

软件工程师捉虫系列



DeBUGGING C++

C++ 程序调试

实用手册

[美] Chris H.Pappas 著
William H.Murray, III
段来盛 郝阿朋 郝曙光 译

Mc
Graw
Hill

McGraw-Hill
<http://www.mhhe.com>



电子工业出版社
Publishing House of Electronics Industry
URL: <http://www.phei.com.cn>

软件工程师捉虫系列



DeBUGGING C++

C++ 程序调试

实用手册

作为一个软件工程师，如果你仍然在为不能捉尽自己开发的C++应用程序中的“虫子”而苦恼不已，那么请你认真地阅读这本书。只要你确实掌握了书中所论述的思想、策略、技术和方法，那么开发无错误的C++应用程序就是既可望也可及的事了。

- ▲ 编写好的代码
- ▲ 使用编译器优化
- ▲ 逻辑与语法错误
- ▲ debugger
- ▲ 调试版本与发行版本
- ▲ 定位、分析和修复命令行代码错误
- ▲ 调试内联汇编语言代码
- ▲ 在 Windows 代码中定位、分析和修复错误
- ▲ 定位、分析和修复命令行中的错误
- ▲ 使用 MFC 类库开发 Windows 程序
- ▲ 定位、分析和修复 MFC Windows 代码中的错误
- ▲ STL 编程实践
- ▲ 定位、分析和修复 STL 代码中的错误
- ▲ 使用 DLL 工作
- ▲ 使用 ActiveX 控件工作
- ▲ 调试 COM、ATL 和 DHTML
- ▲ STL 和 MFC 编程

分 类：软件调试
内 容：C++ 程序调试

读者定位：



初级 中级 高级



天 启 星 责任编辑：寇国华
First Star 封面设计：赵冀江

ISBN 7-5053-6214-3



9 787505 362147 >

ISBN 7-5053-6214-3/TP·3349

定价：56.00 元



C++ 程序调试 实用手册

[美] Chris H. Pappas 著
William H. Murray, III

段来盛 郝阿朋 郝曙光 等译

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 提 要

本书作为有关 Visual C++ Debugger 的专著,是一本非常难得的好书。书中深入地分析了开发不包含逻辑和语法错误的代码技巧以及调试程序的基本原理,介绍了开发和调试命令行代码的过程和方法,说明了关于定位、分析及修复编程错误的方法,介绍了开发 Visual C++ 程序时所遇到的特殊调试问题。

本书是所有软件工程师的必读书籍,也可作为大专院校师生的参考资料。

Chris H.Pappas, William H.Murray, III : DeBUGGING C++ ,first Edition

ISBN 0-07-212519-5

Copyright © 2000 by the McGraw-Hill Companies, Inc.

Authorized translation from the English language edition published by McGraw-Hill Inc.

All rights reserved.

本书中文简体字版由电子出版社和美国麦格劳-希尔国际公司合作出版。未经出版者书面许可,不得以任何方式复制或抄袭本书的任何部分。

版权所有,翻印必究。

图书在版编目(CIP)数据

C++ 程序调试实用手册/(美)帕颇斯(Pappas.C.H.)等著;段来盛等译.

—北京:电子工业出版社,2000.9

(软件工程师捉虫系列)

ISBN 7-5053-6214-3

I. C... II. ①帕...②段... III. C 语言-程序设计-手册, IV. TP. 319

中国版本图书馆 CIP 数据核字(2000)第 71277 号

丛 书 名: 软件工程师捉虫系列
书 名: C++ 程序调试实用手册
原 书 名: DeBUGGING C++
著 者: [美] Chris H.Pappas William H.Murray, III
译 者: 段来盛 郝阿朋 * 郝曙光 等
责任编辑: 寇国华
印 刷 者: 北京天竺颖华印刷厂
出版发行: 电子工业出版社 URL: <http://www.phei.com.cn>
北京市海淀区万寿路 173 信箱 邮编 100036
经 销: 各地新华书店
开 本: 787×1092 1/16 印张: 33 字数: 688 千字
版 次: 2000 年 10 月第 1 版 2000 年 10 月第 1 次印刷
定 价: 56.00 元
书 号: ISBN 7-5053-6214-3/TP · 3349

著作权合同登记号 图字: 01-2000-2008

凡购买电子工业出版社的图书,如有缺页、倒页、脱页、所附磁盘或光盘有问题者,请向购买书店调换。若书店售缺,请与本社发行部联系调换。电话 68279077

译者的话

本书是有关 Visual C++ Debugger 的专著，是所有 Visual C++ 程序员都应该认真阅读的一本非常难得的好书。

作为 Visual C++ 程序员，我们在每天的编程实践中都会遇到各种各样的编程错误，包括语法错误和逻辑错误。当然，我们都知道 Microsoft 提供了一个强有力的调试工具，即 Debugger。但是大多数程序员都没有真正地理解和使用过这一工具，或者即使是使用了这一工具，也只是使用了其中最简单的一些功能。当遇到编码错误时，我们通常的做法是，按照我们个人已经习惯的传统“土”方法调试。这使得编程调试工作变得艰难而乏味，工作效率低下，而所编写的软件产品的效率和稳定性也很差。那么，我们为什么不使用 Debugger 这一强有力的工具呢？原因可能是多方面的，或许是由于习惯，主观上不愿意掌握和使用这一工具，或许是由于图书市场上缺乏有关 Debugger 的专门介绍，客观上导致大多数程序员没有能够很好地理解和掌握这一工具。

目前图书市场上的有关 Visual C++ 图书很多，但大多数只是使用一到两章的篇幅，概括性地介绍 Debugger 的一些主要功能。本书则恰恰相反，它并不是一本有关编程的书籍，而是关于 Visual C++ Debugger 的一本专著。本书通过大量的程序实例，从分析优化器的原理出发，由浅入深地全面介绍了与 C++ Debugger 相关的所有知识。概括起来本书有如下几个方面的特点：

1. 从根本上分析介绍了开发好的代码的编程技巧。包括优化编译器的工作原理，不包含逻辑和语法错误的强壮代码的概念，以及调试程序的基本原理。
2. 全面地介绍了开发和调试命令行代码的过程和方法。包括内联汇编语言代码的方法和面向过程的 Windows 编程中所产生的编程问题，面向对象的命令行代码和 MFC Windows 编程中的编程问题。并给出了使用 Debugger 和其他软件工具，解决一些有趣的实际代码错误的例子。
3. 深入地探讨了标准模板库 (Standard Template Library, STL)，讨论了各种 STL 编码实践，介绍了关于定位、分析及修复编程错误的方法。
4. 详细地介绍了处理 DLL、ActiveX、COM 及使用 STL 和 MFC 开发的程序时所遇到的特殊调试问题。

通过学习本书，读者必定可以对 C++ 编译、优化及调试原理有进一步的认识和理解，全面掌握 C++ Debugger 的各种调试方法和技巧，使编程效率得到长足的提高，使所编写的程序更加强壮。

本书原著语言生动，描述清晰，由于译者水平有限，译稿中难免有不妥之处，望读者批评指正。

本书第 1 章到第 8 章由段来盛同志翻译，第 9 章到第 13 章由郝阿朋同志翻译，第 14 章到第 17 章由郝曙光同志翻译。另外，参加本书翻译整理工作的还有：韩珊、李鑫、于书举、张翔、孙春来、官章全以及侯国峰等同志，在此表示衷心感谢。

译者
2000 年 9 月

前 言

1999 年秋天，笔者与 Osborne/McGraw-Hill 的编辑洽谈关于一本书的建议，这本书从根本上突破了通常的编程图书。这一想法是编写一本通过广泛的编程方案，集中于实际的编程问题，解释 Visual C++ Debugger 的使用方法的书籍。

虽然本书中包含了大量的程序，但它并不是一本编程书籍。本书并非像大多数参考大全(包括笔者本人的那些书)一样，使用一到两章的篇幅介绍 C++ Debugger，而是带领读者进入程序员的思维世界。

首先，我们将介绍如何开发好的代码编写技巧。在第 1 章到第 5 章中，我们将学习优化编译器如何工作，以帮助程序员生成无逻辑和语法错误的强壮代码，还将学习调试程序的基本原理。

在第 6 章到第 8 章中，我们将学习如何开发和调试命令行代码。作为用 C++ 实现较为困难的编程结果的方法，介绍了内联汇编语言代码的方法。最后，我们将简单介绍在面向过程的 Windows 编程中所产生的编程问题。

在第 9 章到第 11 章中，我们将使用调试器进入激动人心的面向对象的编程世界，这一部分彻底检查了命令行代码和 MFC Windows 代码。在第 11 章中，我们将看到使用 Debugger 和其他软件工具解决一些有趣的实际代码错误的例子。

第 12 章和第 13 章说明了 Standard Template Library(STL)，讨论了各种 STL 编码实践，并介绍了关于定位、分析及修复编程错误的方法。

第 14 到第 17 章则专门介绍了处理 DLL、ActiveX、COM 及使用 STL 和 MFC 开发的程序时所遇到的那些特殊调试情况。

与我们一同研究所有可以在 Visual C++ 环境中给予程序员帮助的工具，与我们一同进入 C++ 程序员的思维世界吧。

本书所使用的约定

在几乎每一章中都包括有如下三种特殊的要素：

24x7

这一文本用于指定编程实践，可以采用这些方法保证所编写的代码强壮、稳定，并可使应用程序保持长达一周 7 天，每天 24 小时。

错误监视

这一标志说明可能包含逻辑或语法错误的程序。

设计提示

这一标志将指导读者实现更好的编程实践，帮助读者避免开发坏编码的习惯。设计提示通常引导读者绕过设计低劣但可以工作的代码。

说明：本书中所使用的所有代码都可以从 www.osborne.com 下载获得。

目 录

第一部分 代码开发技巧

第 1 章 编写好的代码	3
1.1 谁需要本书?	4
1.2 教学方法	5
1.3 从何处开始阅读?	5
1.4 警告! 并非所有的 C/C++ 编译器都完全相同.....	6
1.5 语言无关的设计工具 101	6
1.5.1 准备	7
1.5.2 模型!	7
1.5.3 结构图、伪代码及 IPO 框图	8
1.6 好的程序设计的五点要素	12
1.7 规则意味着可以打破	13
1.7.1 安塞尔亚当斯(Ansel Adams)或毕加索(Picasso)	13
1.7.2 注释块	13
1.7.3 标识符: identifiers、IDENTIFIERS 和 Identifiers	14
1.7.4 间隔与缩进	15
1.8 数据类型	20
1.9 匈牙利表示法	23
1.9.1 MFC、句柄、控件及结构的命名规范	23
1.9.2 一般前缀命名规范	24
1.9.3 变量命名规范	24
1.9.4 应用程序符号命名规范	25
1.9.5 Microsoft MFC 宏命名规范	25
1.9.6 库标识符命名规范	26
1.9.7 静态库版本命名规范	26
1.9.8 动态连接库命名规范	26

1.9.9 windows.h 命名规范	26
1.10 操作符优先级	27
1.11 小结	28
第 2 章 使用编译器优化	31
2.1 编码的责任与编译器的优化	32
2.2 Microsoft Visual C++ 的优化	33
2.2.1 调度指令	34
2.2.2 函数级连接	34
2.2.3 字符串池	35
2.2.4 使用 register 键字	35
2.2.5 常量和复制的传播	36
2.2.6 消除死代码和死存储	37
2.2.7 删除冗余子表达式	37
2.2.8 优化循环	37
2.2.9 降低强度	38
2.2.10 inline 键字的使用	38
2.2.11 省略帧指针	39
2.2.12 关闭堆栈检查	39
2.2.13 覆盖堆栈	40
2.2.14 函数调用之间允许使用别名	41
2.2.15 全局优化	41
2.2.16 产生内部函数的内联	41
2.2.17 优化 math.h	42
2.3 Microsoft C++ 的优化开关	42
2.4 使用 Microsoft Visual Studio 设置编译器选项	43
2.4.1 Project Settings 对话框中的 General 类型	45
2.4.2 Project Settings 对话框中的 Code Generation 类型	46
2.4.3 选择结构对齐方式	48
2.4.4 Project Settings 对话框中的 Customize 类型	48
2.4.5 Project Settings 对话框中的 Optimizations 类型	48
2.5 建立发行版本的建议	51
2.6 小结	51

第 3 章 逻辑与语法错误	53
3.1 好的调试策略	55
3.2 四种程序错误类型	56
3.2.1 语法错误	56
3.2.2 连接错误	57
3.2.3 运行错误	58
3.2.4 逻辑错误	60
3.3 查看错误消息	60
3.4 预防性维护	62
3.4.1 桌面检查的含义	63
3.5 异常处理设计	63
3.6 “请多多支持”	64
3.7 Microsoft Visual C++的帮助	65
3.8 小结	66
第 4 章 debugger	67
4.1 确认 Debugger 可以使用	68
4.2 启动 Debugger	69
4.2.1 Step Into 和 Step Over 的区别	71
4.2.2 Go	71
4.2.3 Run to Cursor	71
4.3 理解 Debugger 工具栏图标	72
4.3.1 Restart	73
4.3.2 Stop Debugging	73
4.3.3 Break Execution	73
4.3.4 Apply Code Changes、Edit and Continue	73
4.3.5 Show Next Statement	74
4.3.6 Step Into	74
4.3.7 Step Over	75
4.3.8 Step Out	75
4.3.9 Run to Cursor	75
4.3.10 QuickWatch	75
4.3.11 Watch	75

4.3.12	Variables.....	75
4.3.13	Registers.....	76
4.3.14	Memory.....	76
4.3.15	Call Stack	76
4.3.16	Disassembly	76
4.3.17	Debugger Toolbar Menu Equivalents	76
4.4	其他 Debug 菜单选项.....	77
4.4.1	Step Into Specific Function.....	77
4.4.2	Exceptions.....	77
4.4.3	Threads.....	77
4.4.4	Modules.....	77
4.5	本地菜单 Debugger 选项.....	77
4.5.1	List Members	77
4.5.2	Type Info	78
4.5.3	Parameter Information	78
4.5.4	Complete Word	78
4.5.5	Go To Definition/Reference.....	79
4.5.6	Go To Disassembly	79
4.5.7	Insert/Remove Breakpoint	79
4.6	Debugger 窗口	79
4.6.1	Trace 窗口	80
4.6.2	Watch 窗口	80
4.7	View 菜单和 Debugger 窗口	80
4.7.1	Workspace	80
4.7.2	Output.....	81
4.8	以不同的数据类型查看观察变量.....	81
4.9	打开 Just-in-Time 调式.....	83
4.10	Options 窗口中的 Debug 标签	83
4.10.1	Hexadecimal Display	84
4.10.2	Source Annotation.....	84
4.10.3	Code Bytes	84
4.10.4	Symbols.....	84
4.10.5	Parameter Values.....	84
4.10.6	Parameter Types.....	84

4.10.7	Return Value.....	84
4.10.8	Load COEF & Exports.....	84
4.10.9	Address.....	84
4.10.10	Format	84
4.10.11	Re-evaluate Expression.....	85
4.10.12	Show Data Bytes.....	85
4.10.13	Fixed Width.....	85
4.10.14	Display Unicode Strings	85
4.10.15	View Floating Point Registers.....	85
4.10.16	Just-in-Time Debugging.....	85
4.10.17	OLE RPC Debugging	85
4.10.18	Debug Commands Invoke Edit and Continue	85
4.11	键盘映射	86
4.12	Debugger 快捷键	87
4.13	小结	87

第 5 章 调试版本与发行版本..... 89

5.1	缺省的调试版本建立与发行版本建立设置.....	90
5.2	为调试版本建立修改工程设置.....	90
5.2.1	修改调试选项.....	91
5.2.2	修改产生调试信息的格式.....	91
5.2.3	产生一个映射文件.....	91
5.2.4	重定向调试输入和输出.....	92
5.3	什么是.pdb 文件?	92
5.4	什么是.dbg 文件.....	93
5.5	调试优化的代码.....	93
5.6	打开 Debugger 的另一种方法.....	96
5.7	使用基本版或调试版本.....	96
5.8	C/C++运行调试库.....	97
5.8.1	旧版 iostream.h 和新版 iostream 之间的混乱.....	98
5.9	连接器参考资料	99
5.10	在调试版本中检测发行版本错误.....	101
5.10.1	局部变量的自动初始化.....	101
5.10.2	检查函数指针调用堆栈的合法性	102

5.10.3 检查调用堆栈的合法性	102
5.11 TRACE 宏	102
5.12 VERIFY 宏	103
5.13 移植 Visual C++ 旧的 32 位版本	103
5.13.1 转换早期的 32 位工作空间和工程	104
5.13.2 与 Visual C++ 以前的版本共存	104
5.14 小结	105

第二部分 面向过程的环境

第 6 章 定位、分析和修复命令行代码错误 109

6.1 快速启动调试	110
6.1.1 启动 Debugger 的快速方法	112
6.1.2 变量初始化跟踪	115
6.1.3 小心调试代码	121
6.1.4 快速查看变量的内容	124
6.1.5 中途停止 Debugger	125
6.1.6 执行到代码的指定行	125
6.1.7 全速执行到一个断点	127
6.1.8 运行至光标处	128
6.1.9 现在测试	128
6.2 高级 Debugger 技巧	129
6.2.1 使用新值运行	130
6.2.2 循环调试技巧	141
6.2.3 调用调试函数	142
6.2.4 递归调用与调用堆栈	149
6.2.5 查看反汇编代码	154
6.3 进一步观察变量	157
6.3.1 使用 QuickWatch 窗口	158
6.3.2 使用 Watch 窗口	159
6.4 小结	160

第 7 章 调试内联汇编语言代码 161

7.1 汇编语言初步	162
------------------	-----

7.1.1	数据类型.....	162
7.1.2	寄存器.....	163
7.1.3	寻址模式.....	163
7.1.4	指针.....	164
7.1.5	协处理器.....	164
7.2	调试.....	165
7.2.1	减法运算.....	166
7.2.2	使用 256 位整数.....	170
7.2.3	程序循环.....	179
7.2.4	使用协处理器求和实数.....	186
7.2.5	使用协处理器计算正切值.....	190
7.3	小结.....	199

第 8 章 在 Windows 代码中定位、分析和修复错误 201

8.1	使用两台计算机调试.....	202
8.1.1	准备远程目标计算机.....	203
8.1.2	准备主计算机.....	204
8.1.3	启动调试会话.....	206
8.2	简明 Windows 入门.....	207
8.2.1	基本的 Windows 代码.....	208
8.2.2	调试文件详述.....	209
8.2.3	程序执行的情况.....	210
8.3	调试.....	217
8.3.1	一个动画位图程序.....	217
8.3.2	使用鼠标绘画.....	233
8.4	小结.....	248

第三部分 面向对象过程的环境

第 9 章 定位、分析和修复命令行中的错误 251

9.1	高级调试工具.....	252
9.1.1	内存卸出.....	252
9.1.2	定位错误参数从何处传递而来.....	255
9.1.3	查找何处修改了指针.....	257

9.2	ClassView 窗口要素	267
9.2.1	ClassView 窗口的 Grouped by Access 功能	269
9.2.2	ClassView 窗口的 Base Classes 功能	270
9.2.3	ClassView 窗口的 References 功能	273
9.2.4	ClassView 窗口的 Derived Classes 功能	273
9.2.5	ClassView 窗口中菜单的其余项	275
9.2.6	ClassView 窗口的 Properties 功能	276
9.2.7	在 ClassView 窗口中添加文件夹	277
9.2.8	在文件夹之间移动类	277
9.2.9	隐藏或显示 ClassView 窗口	277
9.3	调试 argc 和 argv[]	278
9.4	小结	282
第 10 章	使用 MFC 类库开发 Windows 程序	283
10.1	为什么使用类库	284
10.2	一个真正的基础类——CObject	285
10.3	什么是应用程序向导和类向导	287
10.4	一个图形程序	287
10.4.1	使用 AppWizard	288
10.4.2	使用 ClassWizard	294
10.4.3	建立 AppWizard 代码	295
10.4.4	AppWizard 模板代码	296
10.4.5	在客户区的图形对象	304
10.5	剖面法	308
10.6	小结	313
第 11 章	定位、分析和修复 MFC Windows 代码中的错误	315
11.1	内存问题	316
11.1.1	有问题的代码	316
11.1.2	定位和分析	320
11.1.3	修复工程	328
11.2	绘图问题	331
11.2.1	有问题的代码	332
11.2.2	定位和分析	336

11.2.3 修改工程.....	344
11.3 小结.....	348

第四部分 标准模板库 (STL)

第 12 章 STL 编程实践	351
12.1 多体系结构.....	352
12.2 掌握 C++.....	352
12.3 STL——进退维谷的数据结构	354
12.4 初识 STL.....	354
12.5 STL 和 HP 公司.....	354
12.6 大众化的 STL.....	355
12.7 STL 总览.....	355
12.8 ANSI/ISO C++接受 STL 的过程	355
12.9 STL 基本组件.....	356
12.9.1 什么是容器?	356
12.9.2 什么是适配器?	357
12.9.3 什么是算法?	358
12.9.4 什么是迭代器?	358
12.9.5 其他的 STL 组件	359
12.10 完整的 STL 程序包.....	359
12.11 杂乱的 C/C++家族.....	361
12.12 回顾数据结构.....	361
12.12.1 静态与动态.....	361
12.12.2 类型指针.....	362
12.12.3 void 指针.....	363
12.13 复习匈牙利命名法.....	365
12.14 函数重载.....	367
12.15 函数指针.....	367
12.16 运算符重载.....	370
12.16.1 运算符和函数调用的重载	370
12.16.2 编写自己的重载运算符.....	371
12.17 从结构到模板.....	373
12.17.1 template 关键字	374

12.17.2	模板语法.....	374
12.17.3	模板函数.....	375
12.17.4	模板类.....	375
12.18	为什么 STL 比模板好.....	376
12.19	小结.....	377
第 13 章 定位、分析和修复 STL 代码中的错误		379
13.1	从标准 C++ 转向 STL 语法的过程中出现的问题.....	380
13.1.1	用迭代器遍历容器	380
13.1.2	仔细研究迭代器.....	381
13.1.3	流迭代器.....	382
13.1.4	为什么使用 end().....	382
13.1.5	复制列表.....	383
13.1.6	列表中的列表.....	383
13.1.7	STL 字符串指针的麻烦	384
13.1.8	释放 STL 指针	385
13.2	一个 C++ 程序转变为 STL 语法的例子	385
13.2.1	第一步——更新 aSingleCard 类.....	389
13.2.2	第二步——更新 WarDeck 类	392
13.2.3	第三步——修复 STL 代码的执行错误	396
13.2.4	第四步——更新 Opponent 类.....	398
13.2.5	第五步——运转的 STL 程序	399
13.3	STL 语法的源文件 wargame.cpp	400
13.4	小结	405
第五部分 特殊的调试问题		
第 14 章 使用 DLL 工作.....		409
14.1	创建一个基于 MFC 的动态链接库.....	410
14.1.1	头文件 Framer.h	412
14.1.2	源代码文件 Framer.cpp	414
14.1.3	建立 Framer.dll	416
14.2	创建使用 DLL 的主应用程序.....	417
14.2.1	头文件 DLLDemoView.h.....	418

14.2.2	源代码文件 DLLDemoView.cpp.....	419
14.3	更加仔细地查看.....	425
14.3.1	远程调试.....	425
14.3.2	有问题的代码.....	428
14.3.3	改正后的代码.....	432
14.4	小结.....	433
第 15 章	使用 ActiveX 控件工作	435
15.1	开发一个 ActiveX 控件	436
15.1.1	使用 ControlWizard.....	437
15.1.2	Test Container	440
15.1.3	产生一个真实的 Clock 控件.....	443
15.2	调试 Clock 控件	453
15.2.1	准备远程目标计算机.....	453
15.2.2	准备主计算机.....	454
15.2.3	开始调试过程.....	454
15.2.4	查找问题.....	456
15.3	小结	459
第 16 章	调试 COM、ATL 和 DHTML.....	461
16.1	COM 对象模型.....	462
16.2	创建一个 ATL 多边形工程	462
16.2.1	优化模块代码.....	467
16.2.2	测试控件.....	480
16.3	调试 ATL COM 控件	482
16.4	小结	484
第 17 章	STL 和 MFC 编程	485
17.1	产生一个 STL 和 MFC 应用程序	486
17.1.1	复数.....	486
17.1.2	模板语法.....	487
17.1.3	基本的应用程序代码.....	493
17.2	调试.....	497
17.3	小结	505

第一部分

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDE 代码开发技巧

- | | |
|-------|-----------|
| 第 1 章 | 编写好的代码 |
| 第 2 章 | 使用编译器优化 |
| 第 3 章 | 逻辑与语法错误 |
| 第 4 章 | debugger |
| 第 5 章 | 调试版本与发行版本 |



第 1 章

编写好的代码





下面的话意味着什么：“可以轻松地完成这一工作，也可以艰难地完成这一工作”？说到底就是：要尽快地完成我们的程序，并且可靠地运行。由于当今程序的综合需求，加之开发这些程序所使用语言的复杂性，与以前任何时候相比较，都需要我们成为一个更好的程序员。

如果读者已经有 15 年或更长时间的编程经历，则一定知道上述的说法千真万确。没有一个编译器(例如，Borland International 的 Turbo Pascal)可以安装到一张 5 1/4 英寸的软盘上了。相反，一个典型的编译器安装要惊人地吞食掉硬盘的 350 兆字节。就在我们为其强大的新编程工具而赞扬 C 语言的发明者 Dennis Ritchie 和 C++ 语言的发明者 Bjarne Stroustrup 的同时，留给我们的却是掌握这些语言潜能的任务。

当今的开发环境所提供的过多的选项，需要导航才能通过，不论这些开发环境是 Microsoft 的 Visual Studio 组件，Borland International 的 IDE(Integrated Development Environment)，还是基于 Unix 的设计工具，都增加到了掌握 C/C++ 的编程负担。

还有目标环境，也可以说是环境的问题。当今的应用程序为了可以真正地适应市场，必须能够运行于各种体系结构下，并且在许多情况下，必须使用多种人类语言格式(使用 Unicode，16 位情况下为 ANSI ASCII 代码)。是否有人提出过所有这一切都必须在一个多任务 GUI 操作系统下完成呢？

还有，是否有人提出过 ANSI/ISO STL(Standard Template Library)，这一程序库如何标准化公共的编程方案，如向量、堆栈、队列、二进制树等？新的 C/C++ cast 操作符又如何呢？等等，等等...我们知道，即使是在阅读本书时，C 和 C++ 语言也正在成长与进化。

1.1 谁需要本书？

本书既适合于 C 或 C++ 初学者，也适合于对这种新语言已经有多年编程经验者。为什么？因为许多程序员并不知道可以使用这一强有力的调试工具，帮助他们使其程序运行得更快和更精确。

笔者前后共有 15 年的经历，教授学生如何编程。这些学生分散在各个领域，从初学者到有经验的本地大公司的程序员，包括许多我们国家最高机密的军事开发环境。

24x7

所有程序员共有的一个关键因素是缺乏时间，似乎从来没有足够的时间掌握为当今的环境开发程序所涉及的全部组件。

1.2 教学方法

本书的编排非常简单。前面的几章首先复习了所有的代码设计基础知识，这些基础知识将在一个介绍性的 Language Independent Design 课程中讲授给初学者。这一章的标题可以是“一个曾经教授英语(或数学，或科学学，或地质学，或任何与正式编程技巧毫无关联的研究领域)的程序设计讲师从未教授过的所有知识。”

第 2 章到第 5 章解释了如何设置 Microsoft Visual C/C++ Debugger，说明了可以使用的选项。最后，以代码实例讲述了如何使用这些工具调试越来越复杂的算法。首先，我们讲述了如何有效地调试面向过程的程序，然后讲述关于基于对象的技术，最后讲述与新的 ANSI/ISO STL 模板结合的算法。

第 14 章到第 17 章通过检查使用动态连接库(Dynamic Link Libraries, DLLs)、ActiveX 控件、COM 文件以及 STL 和 MFC(Microsoft Foundation Class)库的应用程序，完成了代码实例调试教学。

1.3 从何处开始阅读？

除非读者感到自己是受过一流的“正规”编程教育的少数幸运者之一，否则应该和自己的专业生涯要依赖于它一样地阅读本书的每一页。有人曾说“不知者无过”，但这并不适用于程序员。

下面的十个问题可以证明为什么需要阅读本书的每一页。可以自问一下如下的问题，如果读者确实能够理解其中所包含的每一个概念，那么可以立即转到第 2 章，但是必须诚实。

- 问题 1 作为一个 C/C++ 程序员，你的所有子程序代码都封装在函数体内，你知道还有其他类型的子程序否(与 C/C++ 不同，许多语言还有另外一种类型的子程序称为过程)？
- 问题 2 你是否知道过程子程序和函数子程序之间的所有区别？
- 问题 3 你是否学习过如何解析一个表达式，考虑到操作符的优先级？
- 问题 4 你是否知道术语先行 EOF(look-ahead EOF)和非先行 EOF(non-look-ahead)的含义？
- 问题 5 你是否知道预读(priming read)语句的含义？
- 问题 6 你是否知道标记循环指的是什么？
- 问题 7 你是否知道 static(静态)键字在内层、外层以及作为一个(数据或方法)类成员的作用是什么？
- 问题 8 你是否知道编译器在翻译一个其方法体正式地定义在拥有其类内部的方法和翻译一个其方法体定义在拥有其类之外的方法时的区别？



- 问题 9 你是否知道什么时候操作符 `sizeof()` 函数报告(明显)错误的值及其原因是什么?
 - 问题 10 你是否知道递归算法指的是什么(关于递归的定义请参见递归<grin>)?
- 如果对这些问题中的任何一个不能正确回答,则需要阅读本章的剩余部分。

1.4 警告! 并非所有的 C/C++ 编译器都完全相同

在继续介绍之前,必须首先明确一个概念。本书将以 Microsoft 为视角,讲述如何正确地设计、实现并有效地调试 C 和 C++ 程序!正文中所使用的警告和错误诊断将为我们提供另一个关于 C 和 C++ 语言的视角。由于 Dennis Ritchie 和 Bjarne Stroustrup 在其正式的语言描述中遗留了“灰色区域”,编译器厂商按照其自己的方式填补了遗漏的细节。因此,不应将以后各章中所提供的许多概念应用到其他一些厂商的 C 和 C++ 编译器中。

错误监视

切记,本书中的调试实例以及与之相关的分析和讨论均是关于 Microsoft Visual C++ Debugger 的,Visual C++ 如何警告、诊断以及标记错误是产品特定的。

1.5 语言无关的设计工具 101

有经验的程序员都有一个我们称之为“程序员工具箱”的东西。在这一工具箱中有语言无关的设计教程,有多种语言(汇编、COBOL、Fortran、Pascal、PL/I、Visual Basic 等),有硬件教程,如 Digital Logic、Small Systems Design,当然还有手册类教程和工作经验。

奇怪的是,这一工具箱中最重要的工具之一是其中的第一本教程——语言独立设计工具(Language-Independent Design Tools)。为什么?这是因为这一教程制定了所有过去和现在的语言所基于的基础。即使是面向对象的设计技术,在这一最初的教程中也有其原形。

自问一下这一问题:来自于任何面向对象语言的源文件的一个可执行程序,需要在一个特殊设计的微处理器上运行否?答案当然是否定的。换句话说,无论面向对象的代码看起来是多么的复杂,最终都必须编译和/或翻译为机器代码。所以无论一个可执行程序的源文件是过程性的还是面向对象的,其执行都是在同一个芯片上。

面向对象的语法有什么不同?包装!对象自动化了程序中可能已经执行的操作,但并不给予程序任何附加的原动力。典型情况下,如果一个面向对象的程序执行出现故障,其原因是某些基础程序的设计存在问题。因此,好的代码解决方案的一开始必须首先理解构件块或单个组件。下面的 LIDT 101(Language-Independent Design Tools)讨论引导读者快速浏览一下好的 LIDT 101 教程的内容。是否应该阅读这些内容?是的,如果不阅读这一部分,就如

同是读者想学习如何驾驶飞机，而只是简单地知道了如何打开和关闭自动驾驶仪。

以下采用普通的教学方法讲授一个 LIDT 101 课程，包括对逻辑流程图、简单数据类型、逻辑控制语句、循环、子程序的讨论，以及最后对面向对象的考虑。

1.5.1 准备

选择如下假想情况的原因是其惊人因素。给定一个大多数程序员从不会遇到的编程问题，以证明一个论点。

某一天，你去上学或上班，你勇敢的上司交给你如下任务：Jamie，我要你编写一个可以有效地调度一个具有七条跑道的机场的程序。你的方案必须能够处理所有的紧急情况，包括旅客心脏病突发，跑道中央搁浅了一辆加油车，遭到大风袭击等。你主要的目标是每个小时内安全降落和起飞尽可能多的飞机。

你的反应是：

- 放弃。
- 开始咬自己的指甲，拿起一支香烟，寻找另一个 Nicoderm 斑点，或开始咀嚼五块 Nicorette 口香糖。
- 离开普通咖啡机转向浓咖啡机。
- 立即坐下并开始编写代码。
- 稳健而技术性地将所有好的设计原则应用到一个可靠的解决方案中。

通过设计，这一假定的编程问题并不是任何程序员可以在其头脑中解决的。现在问题变成了程序员将哪些漫无目标的编程杂物与少数包含 to-dos 的 Post-It 注释结合在一起呢？除了完全开发的好的初始设计之外，任何方法都可能潜在导致威胁生命的情况。

1.5.2 模型！

对于一个好的设计方案，最简单、最明显，然而却经常忽略的着手点是建立模型。以时间为假想情景，对于某些编程问题，如果没有计算机解决问题，编写所要使用的逻辑步骤将有效。对于其他的编程问题，最好是建立模型。

虽然这初看起来还是老一套，但对于该机场应用程序的一个好的解决方案，一开始应该是到 Toys “R” Us 的一次旅行，购买七个玩具机场，配有加油车、吊车、飞机及乘务组，然后离开此处前往 Lowe’s，购买风扇、胶水和图钉，最后再到本地的体育用品商店购买一些透明鱼线。

购物狂回来后，你封锁了最关键时刻使用的房间(记住，你需要通过自己的勤奋感动你的上司)，开始建造模型机场。你将飞机放置到跑道上，在机场周围装备上吊车、维护人员、旅客、行李等等。由于这是繁忙机场的一个模型，所以你将鱼线粘贴到一些飞机的顶部，并用图钉将其固定在天花板上，以手动方式模仿飞机的起降。



午饭时间到了，休息片刻后你决定打开风扇，检验大风袭击的效果——好，将整个机场吹离了地板，你认识到使用变速风扇可能是一个更好的注意。有什么问题呢？最初的表现可能是精神错乱，“玩机场游戏”可能实际产生事先没有考虑到的情况。即使最初没有预料到由于风扇功率过强而引起的整个机场毁坏的情况，但这引起你考虑安全操作的机场上出现龙卷风或飓风的情况。如果你没有“玩过机场游戏”，你或许会(或者不会)考虑编写使应用程序在任何条件下都友好所需要的代码。

24x7

专业软件工程师都知道，一个可靠而安全的程序解决方案的根本在于以一个好的设计作为开始，而一个好的设计则以一个实际的模型为开始。

1.5.3 结构图、伪代码及 IPO 框图

建立模型之后，程序解决方案的下一步是使用计算机无关语言表示应用程序将执行的逻辑步骤。从历史上看，这一可视的程序解决方案是使用流程图符号表示的。然而，流程图最适用于汇编语言解决方案(现在很少再用来编写完整的应用程序)和很高级别的层次图。

在这一阶段，层次图一般表示出所有该应用程序和其优先级(谁拥有谁)所需的子程序或算法(黑箱概念)。我们再深入仔细地看看，该可视程序解决方案现在集中于各子程序代码。在这一阶段，可视并且语言无关的结构图最好与伪代码(语法无关逻辑语句)混合使用。下面是结构图符号的一个快速浏览。每一个结构图符号前面都有一个代码实例。

设计提示

使用匈牙利表示法(由 Charles Simonyi 发明)给予程序标识符一个更具意义的名字。匈牙利表示法是在一个标识符的名字前面放置一个助记码或数据类型缩写。例如，整型变量 `NumberOfElements` 使用匈牙利表示法后将变为 `iNumberOfElements`。有关匈牙利表示法更详细的信息在本章稍后提供。

1.5.3.1 内联语句

内联语句以矩形框表示，如下所示：

```
int iNumberOfElements;
```

将这一语句翻译为结构图伪代码如图 1-1 所示。

初始化有效数组项个数的计数器

图 1-1 内联结构图符号



1.5.3.2 条件语句

条件结构图符号表示一个程序的 **if** 或 **if...else** 结构，如下所示：

```
if ( argc = 2 )
    ofstream outputFile ( argv[1] );
else
    ofstream outputFile( "lpt1" );
```

将这一语句翻译为结构图伪代码如图 1-2 所示。

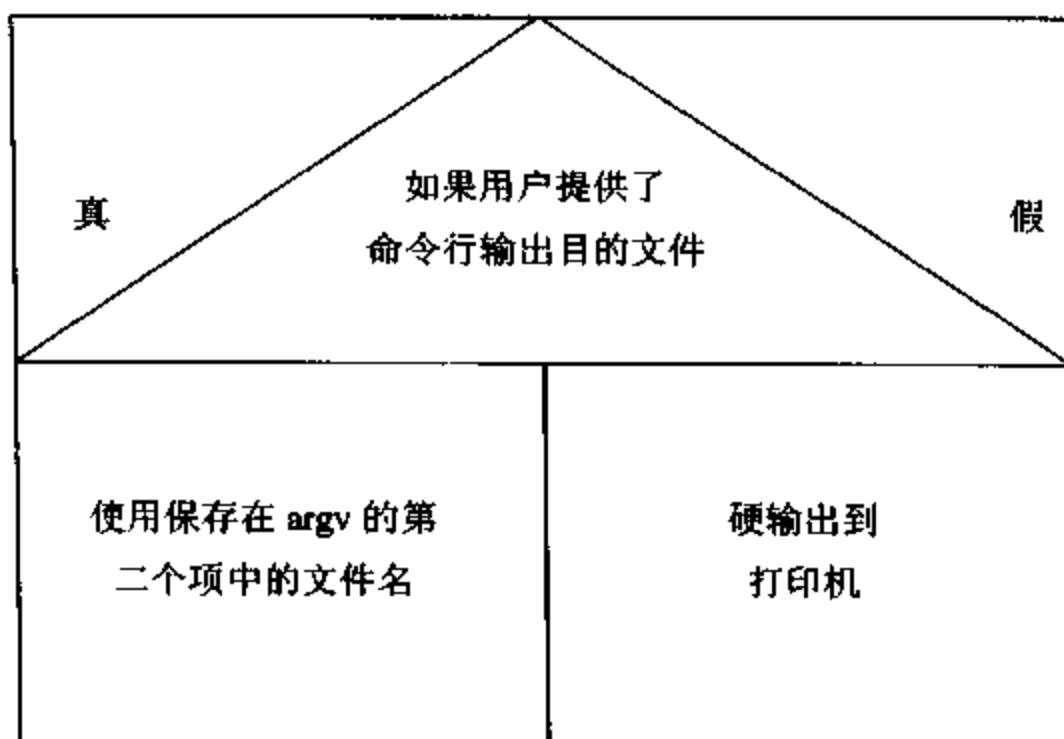


图 1-2 条件结构图符号

1.5.3.3 选择语句

C 和 C++ 使用 **switch...case** 语法表示选择，如下所示：

```
switch( cValueToCategorize )
{
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    case 'y': cVowelCount++;
              break;
    case '.':
    case '!':
```



```
case '?': cSentenceCount++;  
        break;  
default : cConsonantCount++;  
}
```

翻译为结构图形式后如图 1-3 所示。

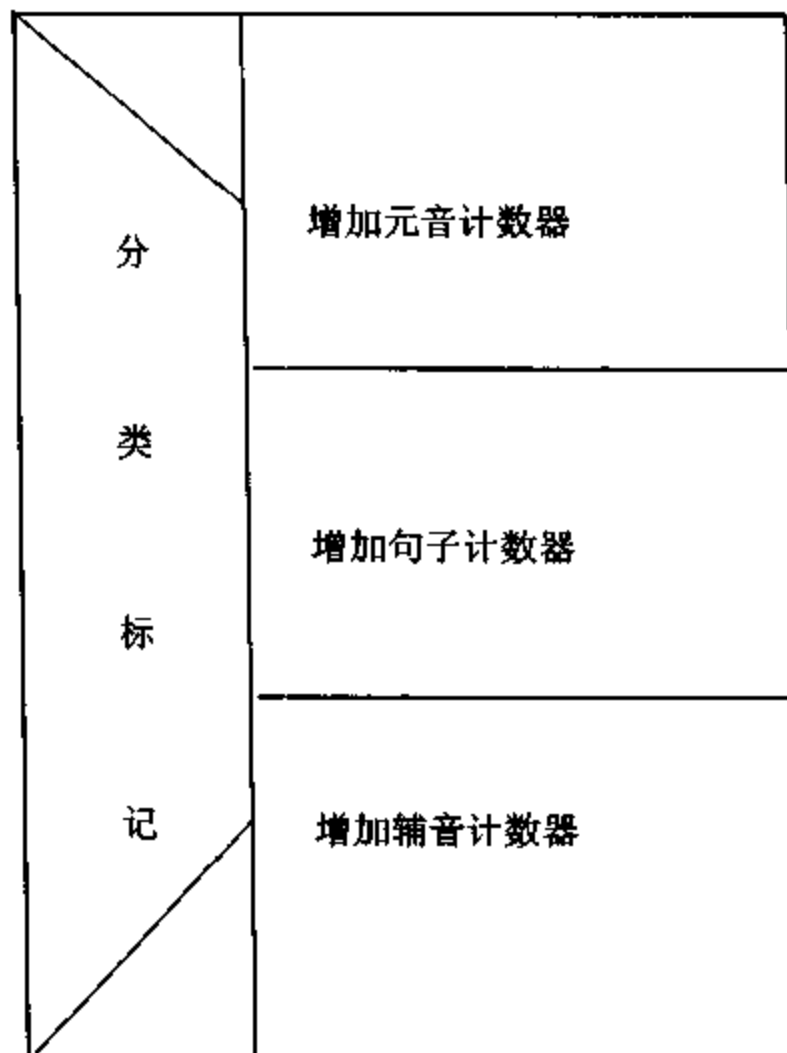


图 1-3 选择结构图符号

1.5.3.4 循环语句

在大多数编程语言中都有两种类型的循环：事先测试循环(**for** 和 **while**)和事后测试循环(**do...while**)，如下所示：

```
for( int rowOffset = 0; rowOffset < MAXELEMENTS; rowOffset++)  
{  
    someArray[ rowOffset ] ...  
}
```

其结构图形式如图 1-4 所示。

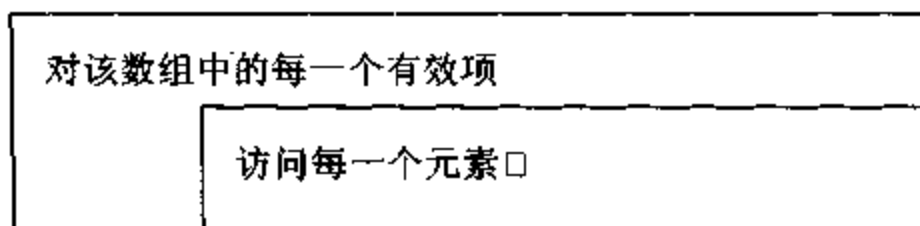


图 1-4 事先测试 for 循环结构图符号

while 循环的结构图符号与事先测试 **for** 循环相同，如下所示：

```
while( ( cValue = cin.get() ) != EOF )
{
    process data...
}
```

翻译为结构图形式后，**while** 循环的结构图符号如图 1-5 所示。

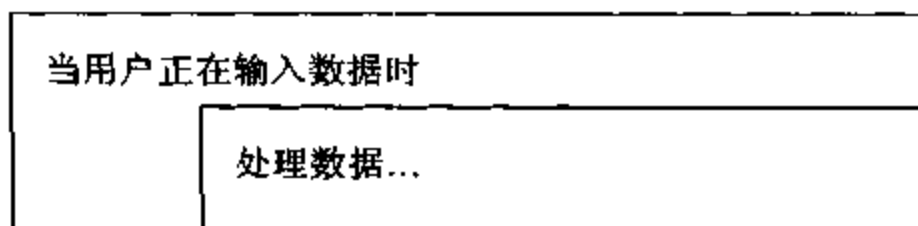


图 1-5 事先测试 while 循环结构图符号

事后测试 **do...while** 循环的结构图符号与事先测试循环恰好相反，如下所示：

```
do
{
    cout << "Company Name\n";
    cout << "\t\tProgram options include:\n";
    cout << "\t\tPress 1 for Online Consultant";
    ...
} while ( (toupper(cMenuSelection = cin.get())) != 'Q')
```

事后测试 **do...while** 循环在结构图中的表示如图 1-6 所示。

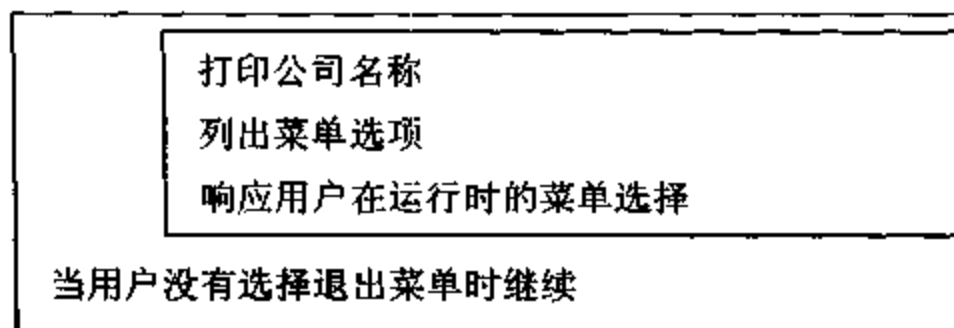


图 1-6 事后测试 do...while 循环结构图符号



1.5.3.5 子程序调用语句

对于函数和方法(成员函数)调用,二者的结构图符号是相同的,即使用一个小斜线语句符号,如下所示:

```
myFunctionCall( myArg1, myArg2, myArg3);  
myMethodCall(myArgA, myArgB, myArgC);
```

其结构图形式如图 1-7 所示。



图 1-7 子程序调用结构图符号

设计提示

对于任何子程序体来说,“单页”(one-page)规则标准是一个非常有用的快速检查方法。这种规则意味着一个算法的编写代码超过一页时就显得太多了,应考虑重新编写该算法——简化其目标。包含这样的代码可能由于过度复杂的逻辑顺序,潜在导致一个设计破坏模块化和升级能力的规则。

结构图与好的伪代码相结合,将使我们很容易地预演一个算法的完整性,不会再有调试这些代码语法的负担。

1.5.3.6 面向对象的语句

由于面向对象的程序在语法结构上重新包装了程序的基础,其结构图符号就是这些程序语言所对应语句的结构图符号的结合物。

1.6 好的程序设计的五点要素

读者可能对问题的解决方案格式即 IPO 图已经熟悉。IPO 图是对由来已久的输入/处理/输出编程问题的一种风格化的处理方法。以下是对这三个基本部分的详细描述,并将整个应用程序开发周期封装起来。所有的程序都必须考虑如下的五部分:

- 从某些输入源获得信息。

- 确定如何排列和保存这一输入。
- 使用一组指令操作这一输入。这些指令可以分为如下四个主要分类：单个语句、条件语句、循环和子程序。
- 报告数据操作的结果。
- 使用好的模型设计、自文档化代码(有意义的变量名等)和合理的缩进方案，综合表示以上各部分。

1.7 规则意味着可以打破

下面的几个小节详细描述了专业人员在使用代码物理表示一个可靠的算法时应该和不应该做的事情。然而，所有的程序员都知道，每一个程序解决方案都有其唯一特定的目标。

除了完成所需的操作外，还需要将应用程序的代码量减小到最少程度。有些应用程序必须使用多年，其设计目标应该是易于修改。而其他一些应用程序解决方案需要为提高执行速度而努力。最后，一个编码后的解决方案也可能作为某些特定产品或特定体系结构特性的指导性实例。虽然所有四种情况都使用了相同的基本原始动力，但每一种情况所提供的算法肯定都有略微的改动。为了实现特殊的目标，努力提高速度的方案甚至可能打破好的代码设计的规则。正是由于这一原因，统一适用于所有情况的一套应该做和不应该做的规则是不存在的。专业程序员能够得到巨额的工资，不仅由于其高水平的设计能力，而且还因为他们所具有的为实现特殊目标使用或不使用代码惯例的技能。

1.7.1 安塞尔亚当斯(Ansel Adams)或毕加索(Picasso)

有经验的软件工程师不仅要在开发其程序解决方案时投入大量的时间，而且在使用代码可视化表示其解决方案时也要投入同样多的时间。真正的专业人员会将其所有的正规训练和经验技巧都专注于每一个所输入的按键上。

实际上这一想法非常简单：将代码页视为一幅画或一张道路图。实现适当的大小写选择(类似于画家对颜色的选择)和缩进方案(类似于艺术家眼中的地平线)，将显著影响底层算法的可理解程度或格调。对大小写和缩进方案不适当的使用，将产生类似于毕加索的画那样的源代码，而对于这些风格工具的正确使用将使另一个程序员很快地诊断和/或了解一个代码段究竟是如何处理的。

1.7.2 注释块

专业的代码编写总是伴随着好的注释。所谓好的注释就是既不致于侮辱一个程序员的智力，如下所示：



```
float fManagersSalary; //fManagersSalary holds the manager's salary
```

也不放过对任何复杂代码的解释。

有一个简单的测试方法可以确定一个代码语句或代码段是否需要注释。假设代码的编写者被安排参加另一个工程，并且几个星期没有看到过他的这一程序。现在，当该程序员回到他的这一应用程序，快速进入到其离开时的位置时，突然间发现自己所查看的语句或代码段使得其脑子一片混乱。换句话说，错综复杂的语法或算法包围了他。这种情况下就需要注释。

设计提示

好的注释既不会侮辱到另一个程序员的智力，也不会显得注释过多。

1.7.3 标识符：identifiers、IDENTIFIERS 和 Identifiers

早期的高级编程语言从来没有和现在的 C/C++ 应用程序那样的完整生命期结构。另外，这些语言中的大部分都没有大小写区分功能，所以标识符的名字如 *mytype*、*MYTYPE* 和 *MyType* 总是被视为同一个名字。

由于当今典型的 Windows 应用程序总是包含有数十个文件，并且在大多数情况下要包含数百个标识符，所以专业人员在生产和布置代码时，几乎需要有一流内部装修工人的技能。除了恰当的注释外，还需要专家水平的大小写使用。

24x7

假设一个正在查看我们的代码的程序员是不区分大小写的，这意味着对他们来说，字符相同而仅仅是大小写不同的多个标识符是同一个变量。然而，我们又转而使用 C/C++ 的大小写区分功能，限制程序员必须跟踪的唯一拼写标识符的个数(因为我们此处假设大小写并不重要)。最终，我们将达到双赢，因为我们建立了一个更容易阅读、跟踪和调试的算法。

一个简单的例子将说明这一点。首先，下面是一个非专业处理方法：

```
typedef struct employees {  
    . . .  
    . . .  
    employees *pToAWorker;  
} personnelRecord;  
void main ( void )  
{  
    personnelRecord employees;  
    . . .  
}
```

让我们分析这个例子，简单的标识符记数：employees、pToAWorker、personnelRecord，

产生总共四个唯一的标识符。考虑可以引起混淆的那些选择，例如，`typedef` 使用标志 `employees` 暗含着复数含义，但该结构将仅仅包含一个单数雇员的记录。

其次，虽然从逻辑上讲该结构的成员 `pToAWorker`，确实指出了该成员将如何使用，但从可读性方面来说，该标识符似乎暗示着它是指向某个其他逻辑概念，即 `AWorker` 的一个指针。最后带有伤害性的代码是对变量 `employees` 的声明，它是一个复数名词，可能暗含有一个结构数组，但此处显然不是。

现在，我们检查使用 C/C++ 的大小写区分功能对上述语句重新进行专业化编写，以保证编译器可以顺畅编译(通过为每一条语法提供一个唯一的大小写字母组合)，同时利用每一个标识符的可读性尽可能地减少程序员的混淆。

```
typedef struct An_Employee {
    . . . .
    . . . .
    An_Employee *pAn_Employee;
} AN_EMPLOYEE;
void main ( void )
{
    AN_EMPLOYEE An_employee;
    . . .
}
```

在上面的这一段代码中唯一拼写标识符的总数只有一个(在阅读匈牙利表示法时，专业程序员将如同 `p...An_Employee` 一样，只观察该标识符的主要部分——`An_Employee`)。这一代码表示极大地改进了唯一标识符的个数(从人类的角度)，更重要的是它减轻了程序员跟踪一个程序所有标识符的负担。

重新编写的代码段对于该结构的标记字段和类型定义名使用了相同的拼写，指针成员除了前面的 `p` 之外也与之匹配。最后，变量 `An_employee` 的类型 `AN_EMPLOYEE` 也从逻辑上将 `typedef` 语句连接到该变量的名字。使用这一重新编写的代码段后，程序员必须完成的全部工作就是理解结构 `An_Employee` 中成员的含义(以大小写的任何组合拼写)，正确使用特定应用程序中的定义、类型或变量。

24x7

C/C++ 软件工程师对于头文件中的定义全部采用大写，对于源文件(.c/.cpp)中的声明采用大小写混合使用方式。唯一的例外是函数原型、类定义及 STL 语法。

1.7.4 间隔与缩进

有时为了编写易于阅读的源代码需要付出一些额外的努力，而这些努力完全是强制性



的。有些情况下，标准的间隔和缩进方案不能在视觉方面区分重要的代码细节。

例如，如下的代码段就语法而言是绝对没有错误的，就编码风格来说也是很典型的。只是其既没有使用大小写方式的匈牙利表示法，也没有根据其含义使用间隔或缩排：

```
#include <iostream>
#define m1 50
#define m2 4
typedef enum cars{yugo, ford, buick, pontiac } carmakes;
typedef struct one_carsale {
    carmakes CarMake;
    char CustomerName[m1],CarModel[m1];
    float Cost;
    one_carsale *NextCarSalePtr;
} acar;
typedef struct salesperson {
    char SalesPersonsName[m1];
    float CommissionRate,BaseSalary,WeeklyPaycheck;
    acar **NextNode;
} aperson;
acar *NextNodePtr[m2];
aperson Sale[m2];
```

现在，设想该代码段要嵌入到数千行的代码中，嵌入到将要运行一二十年的公司的应用程序中。再设想该代码的作者之外的其他一些程序员必须使用这些数据类型、结构及用户定义类型。

重新编写上述同样的语法，将数据类型与标识符名、应用程序特定的枚举类型及其之间的相互关系区分开，如下所示：

```
#include <iostream>
#define MAX_NAMELENGTH 50
#define MAX_SALESPEOPLE 4
/* . . . . . makes of cars being sold */
typedef enum car_makes{ yugo, ford, buick, pontiac } CAR_MAKES;
/* . . . . . structure to represent each car sale */
typedef struct one_carsale {
    CAR_MAKES          enCarMake;
    char               pszCustomerName [ MAX_NAMELENGTH ];
    char               pszCarModel      [ MAX_NAMELENGTH ];
    float              fCost;
    one_carsale        *pstNextCarSalePtr;
} ONE_CARSale;
/* structure to represent each individual sales person */
```

```
typedef struct salesperson {
    char          pszSalesPersonsName[ MAX_NAMELENGTH ];
    float         fCommissionRate,
                 fBaseSalary,
                 fWeeklyPaycheck;
    ONE_CARSALE   **ppstHeaderNode;
} ONE_SALESPERSON;
/* . . . . . declaration for non-standard array-of-pointers */
ONE_CARSALE *apstHeaderNodePtr [ MAX_SALESPEOPLE ];
/* . . . . . array of salespeople */
ONE_SALESPERSON astSalesTeam [MAX_SALESPEOPLE ];
```

此处，程序员可以方便地提出一些重要的问题，使用这种代码风格提高了找到答案的速度。例如，观察这一代码段程序员可以很方便地回答如下的问题：

- 在这一代码段中需要哪些头文件？
- 符号常量的名字是什么？
- 符号常量的代换值是什么？
- 这一应用程序中使用了哪些标准的数据类型？
- 如果有，那么用户定义类型有哪些？
- 成员的名字是什么？
- 如果有，那么数组的维数是多少？
- 这一代码段是否指出了任何关于底层的逻辑假设？
- 这一程序中变量是如何使用的？

从更富有意义的标识符名字——例如 *ml* 变为 *MAX_NAMELENGTH*——到匈牙利表示法 *ppstHeaderNodePtr* (暗示出隐含的双重指针**和成员的逻辑用法——虚拟头标结点 *ptr*，与数据结点 *ptr* 相反)，到间隔的使用，这一例子在其含义方面更加清晰。

符号常量的代换值 50 和 4 被以列的形式表示，结构的定义从视觉上将成员数据类型与成员的名字分割开，并方便了对标量大小描述的观察：

```
typedef struct one_carsale {
    CAR_MAKES      enCarMake;
    char           pszCustomerName [ MAX_NAMELENGTH ];
    char           pszCarModel      [ MAX_NAMELENGTH ];
    float          fCost;
    one_carsale    *pstNextCarSalePtr;
} ONE_CARSALE;
```

在定义同一数据类型的变量时，*ONE_CARSALE* 使用了一次说明一个的方法——此处为 *char*，即每一个串成员都以 *char* 作为其类型占用单独的一行。对一个专业的程序员来说，



这一风格需要程序员对成员的名字给予特别注意。

然而，下一个结构的定义使用了另一种不同的方法。注意下面代码段中定义的 **float** 成员：

```
typedef struct salesperson {  
    char            pszSalesPersonsName[ MAX_NAMELENGTH ];  
    float           fCommissionRate,  
                   fBaseSalary,  
                   fWeeklyPaycheck;  
    ONE_CARSALE     **ppstHeaderNode;  
} ONE_SALESPERSON;
```

在这一例子中，对于所有类型相似的成员(float)只放置了一个数据类型，巧妙地说明了这三个成员之间的逻辑关系。

必须承认，.exe 文件毕竟是一个.exe 文件，如果正在发送的货物只是最终产品，可能并不需要花费大量的时间以这种易于阅读的格式再重新处理源代码。然而，如果一个应用程序具有合理的开发周期，代码风格的重新编写可能将对以后几年模块化的简化有巨大的回报。作为一个对比实例，为产生一个指导性方法，下述程序将一个算法的内容形式化了。在这一形式化的程序中包含了许多专业编码经验。实例代码给出了如何使用 Borland International 的 dos.h 和 conio.h 头文件，通过中断 33h 输出简单文本模式图形后打开鼠标的方法(Windows 操作系统控制的应用程序并不赞成如此处理)。下面是该算法：

```
/* **** */  
/* program to use interrupt 33h for mouse control */  
/* **** */  
  
#include <stdio.h> /* standard C I/O definitions */  
#include <dos.h> /* containing intr() function prototype */  
#include <conio.h> /* simple text mode graphics routines */  
#define FILE_MENU_Y_COORD 1 /* symbolic constants used to delimit */  
#define FILE_MENU_LOW_X_COORD 1 /* the FILE menu item row and starting */  
#define FILE_MENU_HIGH_X_COORD 4 /* starting and ending column positions */  
void main( void )  
{  
    struct REGPACK all_gp_registers; /* REGPACK defined in dos.h */  
    int iButton = 0, iRow, iColumn;  
    clrsc ( );  
    _setcursortype ( _NOCURSOR ); /* conio.h, turn cursor off */  
    textbackground ( BLUE );  
    textcolor ( LIGHTGRAY );  
    gotoxy ( 1,1 );  
    cprintf ( "FILE" ); /* cprintf for color output */
```

```

all_gp_registers.r_ax = 1;                                /*          show pointer */
intr ( 0x33,&all_gp_registers );                          /*          call-by-reference */
while ( iButton == 0 ) {
    iButton = all_gp_registers.r_bx;                      /*          get button status */
    gotoxy ( 30, 10 );
    cprintf ( "%2d Button (0-none, 1-left,2-right)",iButton);
    iColumn = all_gp_registers.r_cx/8+1;                  /*          divisions by 8 convert */
    gotoxy ( 30, 11 );                                    /*          pixels to rows, columns */
    cprintf ( "%2d x coordinate", iColumn );
    iRow      = all_gp_registers.r_dx/8+1;
    gotoxy ( 30, 12 );
    cprintf ( "%2d,y coordinate", iRow );
    if ( ( iRow      == FILE_MENU_Y_COORD ) &&           /*          if mouse row and */
        ( iColumn >= FILE_MENU_LOW_X_COORD ) &&         /*          column coordinates */
        ( iColumn >= FILE_MENU_HIGH_X_COORD ) )         /*          over FILE menu */
        printf( ( "\a" );                                /*          ring bell */
    }                                                       /*          endwhile */
}                                                         /*          end main */

```

为了逻辑上将执行特定任务的语句集中在一起，这一指导性算法使用了行内间隔法。这种方法允许程序员提出如下的问题(自顶向下查看该算法)：

- 这一算法的功能是什么(第一个注释块给出答案)?
- 这一算法需要的头文件有哪些?
- 如果有的话，需要使用的符号常量有哪些?
- 需要哪些变量?
- 初始化屏幕需要使用的代码有哪些?
- 设置屏幕输出选项需要使用的代码有哪些?
- 哪些菜单项为输出项?
- 初始化鼠标中断需要哪些代码?
- 哪些代码用于测试和报告鼠标按钮的按击情况?
- 哪些代码用于处理鼠标报告的列坐标?
- 哪些代码用于处理鼠标报告的行坐标?
- 哪些代码确认鼠标正处于一个菜单项之上?

大量增强性代码风格继续以列化的注释段使用，这使得程序员可以很容易地查看注释和可执行代码。三个**#define**符号常量：

```

#define FILE_MENU_Y_COORD      1    /*          symbolic constants used to delimit */
#define FILE_MENU_LOW_X_COORD  1    /*          the FILE menu item row and starting */
#define FILE_MENU_HIGH_X_COORD 4    /*          starting and ending column positions */

```



设置了该算法的先后次序。该程序的编写使用三个符号常量为每一种程序可能选择的菜单项(该例中仅有“FILE”)显示和检测鼠标。

如果程序员打算显示更多菜单项,则可以为每一个新菜单项统一定义更多的符号常量即可。每三个一组的新符号常量标记出新菜单项的行(Y_COORD)、开始列(LOW_X_COORD)和结束列(HIGH_X_COORD)的屏幕坐标。对应的 if...语句将测试何时所报告的鼠标坐标处于新菜单项之上。

注意,符号常量定义与匹配的 if...语句之间的关系从一开始就设计到了算法中,这就允许使用统一的代码风格在以后插入更多的菜单项显示和检测。

如下所示,列化的实际参数允许程序员方便地查看与当前的参数值相对应,该算法需要有哪些函数:

```
textbackground ( BLUE      );
textcolor      ( LIGHTGRAY );
gotoxy         ( 1,1       );
cprintf        ( "FILE"    );      /* cprintf for color output */
```

最后,由于三个符号常量的定义,下述代码段:

```
if ( ( iRow == FILE_MENU_Y_COORD ) && /* if mouse row and */
      ( iColumn >= FILE_MENU_LOW_X_COORD ) && /* column coordinates */
      ( iColumn <= FILE_MENU_HIGH_X_COORD ) ) /* over FILE menu */
    printf( ( "\a" ); /* ring bell */
```

将这三个符号常量的定义与风格化的条件语句 if...结合起来,使得程序员知道正在与当前鼠标位置比较的菜单项(在该例中为“FILE”)的行/列坐标是哪一个。注意这一风格是如何方便地允许:

- 查看所拥有的控制语句(if...).
- 检查正在检验的菜单项是哪一个(FILE_).
- 计算条件个数(三个).
- 检查组合中使用的逻辑操作符.
- 观察所产生的结果,如果为真的话.

为了在视觉上增强一个算法的页面形式,所需的关于富有含义、风格细节及时间要求的争论是双方面的。然而,职业经验非常简单:使用一种使其他人而不是代码编写者本人容易理解的代码风格。

1.8 数据类型

最简单的程序计算错误源之一是开始就选择了不正确的存储变量的数据类型。对于多平

台支持的 C/C++ 而言，可能如同程序员没有认识到 32 位的 C/C++ 编译器为 `int` 数据类型分配 4 个字节，而 16 位的 C/C++ 编译器只分配 2 个字节一样简单。当然，如果我们正在使用 4 字节整数的最大合法整数值，但是在一个 16 位的环境下运行该程序，则将产生计算错误。

有三种主要手段可以保证程序中不引入这些类型的运行错误。首先，可以打印所使用编译器的 Help 屏幕，仔细查看当前实现的值范围。表 1-1 列出了 C/C++ 的数据类型，可以作为 Microsoft Visual Studio C/C++ 程序员的参考工具。

表 1-1 Microsoft C/C++ 的标准数据类型

类型名称	字节	其他名称	值范围
<code>int</code>	*	signed, signed int	依赖于系统
<code>unsigned int</code>	*	unsigned	依赖于系统
<code>_int8</code>	1	char, signed char	-128 到 127
<code>_int16</code>	2	short, short int, signed short int	-32,768 到 32,767
<code>_int32</code>	4	signed, signed int	-2,147,483,648 到 2,147,483,647
<code>_int64</code>	8	无	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807
<code>char</code>	1	signed char	-128 到 127
<code>unsigned char</code>	1	无	0 到 255
<code>short</code>	2	short int, signed short int	-32,768 到 32,767
<code>unsigned short</code>	2	unsigned short int	0 到 65,535
<code>long</code>	4	long int, signed long int	-2,147,483,648 到 2,147,483,647
<code>unsigned long</code>	4	unsigned long int	0 到 4,294,967,295
<code>enum</code>	*	无	同 int
<code>float</code>	4	无	3.4E+/-38(7 位)
<code>double</code>	8	无	1.7E+/-308(15 位)
<code>long double</code>	10	无	1.2E+/-4932(19 位)

长双精度数据类型(80 位、10 字节精度)直接映射为 Windows NT 和 Windows 95 中的双精度(64 位、8 字节精度)。

`signed`(有符号)和 `unsigned`(无符号)是可以用于任何整数类型的标识符。缺省情况下 `char`(字符)类型为 `signed`，但也可以指定 `_U`，使其缺省为 `unsigned`。

`int` 和 `unsigned int` 类型的大小为系统的字大小。在 MS-DOS 和 16 位版本的 Windows 中为 2 字节(与 `short` 和 `unsigned short` 相同)，在 32 位的操作系统中为 4 字节。然而，可移植的



代码不应该依赖于 int 的大小。

保证程序不引入运行错误的第二种方法包括使用 C/C++ 的 `sizeof()` 操作符。`sizeof()` 是返回一个对象所需内存字节数的操作符(而不是函数)。这一对象既可以是某些内容的一个设计, 如数据类型 `int(sizeof(int))`, 也可以是一个实际变量或对象实例, 如 `int_number_of_scores(sizeof(int_number_of_scores))`。一个简单的测试条件可以为适当的体系结构确定一个代码解决方案, 如下所示:

```
int ivalue;
float fvalue;
if(sizeof(ivalue) == 4)
    ivalue = user_response;
else
    fvalue = user_response;
```

最后一种方法是要了解一个称为 `limits.h` 的特殊标准 C/C++ 头文件, 该文件是为这一用途而特别设计的。作为说明, 下面给出了 `limits.h` 的一部分:

```
#define CHAR_BIT      8           /* number of bits in a char */
#define SCHAR_MIN     (-128)      /* minimum signed char value */
#define SCHAR_MAX     127         /* maximum signed char value */
#define UCHAR_MAX     0xff        /* maximum unsigned char value */

#ifndef _CHAR_UNSIGNED
#define CHAR_MIN       SCHAR_MIN  /* minimum char value */
#define CHAR_MAX       SCHAR_MAX  /* maximum char value */
#else
#define CHAR_MIN       0
#define CHAR_MAX       UCHAR_MAX
#endif /* _CHAR_UNSIGNED */

#define MB_LEN_MAX     2           /* max. # bytes in multibyte char */
#define SHRT_MIN       (-32768)    /* minimum (signed) short value */
#define SHRT_MAX       32767       /* maximum (signed) short value */
#define USHRT_MAX      0xffff      /* maximum unsigned short value */
#define INT_MIN        (-2147483647 - 1) /* minimum (signed) int value */
#define INT_MAX        2147483647   /* maximum (signed) int value */
#define UINT_MAX       0xffffffff  /* maximum unsigned int value */
#define LONG_MIN       (-2147483647L - 1) /* minimum (signed) long value */
#define LONG_MAX       2147483647L   /* maximum (signed) long value */
#define ULONG_MAX      0xffffffffUL /* maximum unsigned long value */

#if _INTEGRAL_MAX_BITS >= 8
#define _I8_MIN        (-127i8 - 1) /* minimum signed 8 bit value */
#define _I8_MAX        127i8        /* maximum signed 8 bit value */
#define _UI8_MAX       0xffui8      /* maximum unsigned 8 bit value */
#endif

#if _INTEGRAL_MAX_BITS >= 16
#define _I16_MIN       (-32767i16 - 1) /* minimum signed 16 bit value */
#define _I16_MAX       32767i16        /* maximum signed 16 bit value */
#define _UI16_MAX      0xffffui16      /* maximum unsigned 16 bit value */
#endif
```

```
#if _INTEGRAL_MAX_BITS >= 32
#define _I32_MIN (-2147483647i32 - 1) /* minimum signed 32 bit value */
#define _I32_MAX 2147483647i32 /* maximum signed 32 bit value */
#define _UI32_MAX 0xffffffffui32 /* maximum unsigned 32 bit value */
#endif
```

1.9 匈牙利表示法

匈牙利表示法(Hungarian Notation)首次由 Microsoft 的 Charles Symonyi 引入, 它通过在每一个标识符名字的前面放置一个数据类型助记符, 为程序员提供了对标识符类型的即时识别。这种内联方式的声明大大增强了解释代码语句的效率, 无须再返回到各标识符的定义处。

首先, 请看下面的典型标识符语法:

```
Value1 = 1;
Value2 = 2;
Result = Value1 / Value2; //Result assigned 0
```

这一例子丝毫没有转达关于变量类型或任何潜在的底层逻辑错误的信息。现在, 请看使用匈牙利表示法重新编写后的如下同等代码:

```
iValue1 = 1;
iValue2 = 2;
fResult = iValue1 / iValue2; //fResult assigned 0 not 0.5
```

此处, 浮点型变量的前置助记符 *f* 和整型变量的助记符 *i*, 为检查该代码语句的任何程序员标记了一个潜在的切断错误。在这一计算中潜在地隐藏丢失精度的情况, 因为 C/C++ 的除法操作符/, 在处理两个整型数值时返回一个整型结果。

变量的名字 *fResult* 暗示该代码的编写者所希望的是一个浮点结果。当匈牙利表示法统一应用于整个应用程序时, 许多这种类型的潜在代码错误将在不顺眼的代码行上立即被肉眼诊断出来。

对一个使用了数十个数据类型、常量(使用 **const** 关键字)、符号常量(使用 **#define** 语法)、结构、类、消息类型等的典型 C++ Windows 应用程序, 匈牙利表示法为程序员免除了总是要搜索嵌入的头文件 **#include**, 查看源代码定义的负担。

表 1-2 到表 1-10 详细描述了 Microsoft 标准化 Windows 定义的方法, 应该仔细阅读这些内容。

1.9.1 MFC、句柄、控件及结构的命名规范

对于 MFC 和 Windows 句柄、控件及结构之间的关系, Microsoft 使用此处列出的命名规范:



Windows 类型	样本变量	MFC 类	样本对象
HWND	hWnd;	CWnd*	pWnd;
HDLG	hDlg;	CDialog*	pDlg;
HDC	hDC;	CDC*	pDC;
HGDIOBJ	hGdiObj;	CGdiObject*	pGdiObj;
HPEN	hPen;	CPen*	pPen;
HBRUSH	hBrush;	CBrush*	pBrush;
HFONT	hFont;	CFont*	pFont;
HBITMAP	hBitmap;	CBitmap*	pBitmap;
HPALETTE	hPalette;	CPalette*	pPalette;
HRGN	hRgn;	CRgn*	pRgn;
HMENU	hMenu;	CMenu*	pMenu;
HWND	hCtl;	CState*	pState;
HWND	hCtl;	CButton*	pButton;
HWND	hCtl;	CEdit*	pEdit;
HWND	hCtl;	CListBox*	pListBox;
HWND	hCtl;	CComboBox*	pComboBox;
HWND	hCtl;	CScrollBar*	pScrollBar;
HSZ	hszStr;	CString	pStr;
POINT	pt;	CPoint	pt;
SIZE	size;	CSize	size;
RECT	rect;	CRect	rect;

1.9.2 一般前缀命名规范

表 1-2 中列出了 Microsoft 使用的一般前缀命名规范。

表 1-2 一般前缀命名规范

前缀	类型	实例
C	类或结构	CDocument, CPrintInfo
m_	成员变量	m_pDoc, m_nCustomers

1.9.3 变量命名规范

表 1-3 列出了 Microsoft 专门用于 Windows 变量的命名规范。

表 1-3 变量前缀命名规范

前缀	类型	描述	实例
ch	char	8 位字符	chGrade
ch	TCHAR	如果_UNICODE 定义, 为 16 位字符	chName
b	BOOL	布尔值	bEnabled
n	int	整型(其大小依赖于操作系统)	nLength



(续表)

前缀	类型	描述	实例
n	UNIT	无符号值(其大小依赖于操作系统)	nLength
w	WORD	16 位无符号值	wPos
l	LONG	32 位有符号整型	lOffset
dw	DWORD	32 位无符号整型	dwRange
p	*	指针	pDoc
lp	FAR*	远指针	lpDoc
lpsz	LPSTR	32 位字符串指针	lpszName
lpsz	LPCSTR	32 位常量字符串指针	lpszName
lpsz	LPCTSTR	如果_UNICODE 定义, 为 32 位常量字符串指针	lpszName
h	handle	Windows 对象句柄	hWnd
lpfn	callback	指向 CALLBACK 函数的远指针	lpfnAbort

1.9.4 应用程序符号命名规范

Microsoft 在其 Windows 应用程序中使用表 1-4 中所列出的符号常量前缀。

表 1-4 Windows 应用程序所使用的符号的命名规范

前缀	符号类型	实例	范围
IDR_	不同类型的多个资源共享标识	IDR_MAINFRAME	1 到 0x6FFF
IDD_	对话框资源	IDD_SPELL_CHECK	1 到 0x6FFF
HIDD_	对话框资源的 Help 上下文	HIDD_SPELL_CHECK	0x20001 到 0x26FF
IDB_	位图资源	IDB_COMPANY_LOGO	1 到 0x6FFF
IDC_	光标资源	IDC_PENCIL	1 到 0x6FFF
IDI_	图标资源	IDI_NOTEPAD	1 到 0x6FFF
ID_	来自菜单项或工具栏的命令	ID_TOOLS_SPELLING	0x8000 到 0xDFFF
HID_	命令 Help 上下文	HID_TOOLS_SPELLING	0x18000 到 0x1DFFF
IDP_	消息框提示	IDP_INVALID_PARTNO	8 到 0xDFFF
HIDP_	消息框 Help 上下文	HIDP_INVALID_PARTNO	0x30008 到 0x3DFFF
IDS_	串资源	IDS_COPYRIGHT	1 到 0x7FFF
IDC_	对话框内的控件	IDC_RECALC	8 到 0xDFFF

1.9.5 Microsoft MFC 宏命名规范

表 1-5 列出了 MFC 宏前缀, 它可以帮助程序员很快理解一个宏的分类。

表 1-5 Microsoft MFC 宏的命名规范

名称	类型
_AFXDLL	唯一的动态连接库(Dynamic Link Library, DLL)版本
_ALPHA	仅编译 DEC Alpha 处理器



(续表)

名称	类型
_DEBUG	包括诊断的调试版本
_MBCS	编译多字节字符集
_UNICODE	在一个应用程序中打开 Unicode
AFXAPI	MFC 提供的函数
CALLBACK	通过指针回调的函数

1.9.6 库标识符命名规范

Microsoft 的库使用表 1-6 中所列出的命名规范。

表 1-6 库命名规范

指示符	值和含义
u	ANSI(N)或 Unicode(U)
d	调试或发行: D=调试; 忽略指示符为发行

1.9.7 静态库版本命名规范

表 1-7 列出了 Microsoft 专用的库命名规范。

表 1-7 静态库版本

库	描述
NAFXCWD.LIB	调试版本: MFC 静态连接库
NAFXCW.LIB	发行版本: MFC 静态连接库
UAFXCWD.LIB	调试版本: 具有 Unicode 支持的 MFC 静态连接库
UAFXCW.LIB	发行版本: 具有 Unicode 支持的 MFC 静态连接库

1.9.8 动态连接库命名规范

表 1-8 列出了 Microsoft 动态连接库或 DLL 所使用的命名规范。

表 1-8 DLL 宏命名规范

名称	类型
_AFXDLL	唯一的动态连接库(DLL)版本
WINAPI	Windows 所提供的函数

1.9.9 windows.h 命名规范

表 1-9 详细描述了 windows.h 所使用的 Microsoft 新的标准类型。这些组合类型对于任何给定的应用程序可能包含不同种类的数据。更详细的列表可使用 Visual C++ Help 工具,

并执行“Win32 Simple Data Types”索引搜索。

表 1-9 windows.h 所使用的命名规范

类型定义	描述
WINAPI	使用在 API 声明中的 FAR PASCAL 位置。如果正在编写一个具有导出 API 入口点的 DLL，则可以在自己的 API 中使用该类型
CALLBACK	使用在应用程序回叫例程如窗口和对话框过程中的 FAR PASCAL 位置
LPCSTR	与 LPSTR 相同，只是 LPCSTR 用于只读串指针，其定义类似于(const char FAR*)
UNIT	可移植的无符号整型类型，其大小由主机环境决定(对 Windows NT 和 Windows 95 为 32 位)。它是 unsigned int 的同义词，使用在 WORD 位置，在一个 32 位的平台上希望得到一个 16 位无符号值的极少数情况除外
LRESULT	窗口程序返回值的类型
LPARAM	声明 lParam 所使用的类型，lParam 是窗口程序的第四个参数
WPARAM	声明 wParam 所使用的类型，wParam 是窗口程序的第三个参数(一种组合数据类型)
LPVOID	一般的指针类型，与(void *)等同。应该使用它来代替 LPSTR

1.10 操作符优先级

程序错误的另一个来源是不正确地选择和/或放置 C/C++的操作符。为了便于代码检查，表 1-10 从最高级到最低级，列出了标准 C/C++的操作符优先级，包括其优先级和结合性。

在一个流行的电视商务节目中提出了这样一个问题：“你是愿意购买车辆保险，还是愿意看牙医？”是的，编写好的代码或许可以与纺丝线相比：我们知道应该这样做，但这样做要花费额外的时间，我们可能不想花费这些额外的时间，但是这样做会有丰厚的回报。一个清晰、可靠、设计精细的算法会光芒四射、光彩照人，它将具有很好的执行效率、可视的用户界面以及数据完整性，这些均将使每一个人的脸上绽放出微笑，从老板到同事，到最终用户。

表 1-10 C/C++操作符优先级

操作符	含义	结合性
最高级		
++	后自增	左到右
--	后自减	
()	函数调用	
[]	数组元素	
->	指向结构成员	
.	结构或联合的成员	
++	先自增	右到左
--	先自减	
!	逻辑 NOT	



(续表)

操作符	含义	结合性
~	位 NOT	
-	一元减	
+	一元加	
&	地址	
*	间接	
sizeof	字节数	
new	分配程序内存	
delete	释放程序内存	
(type)	类型强制[例如, (float)i]	
*	指向成员(对象)	左到右
->*	指向成员(指针)	
*	乘	左到右
/	除	
%	求余	
+	加	左到右
-	减	
<<	左移(LEFT-SHIFT)	左到右
>>	右移(RIGHT-SHIFT)	
<	小于	左到右
<=	小于等于	
>	大于	
>=	大于等于	
=	等于	左到右
!=	不等于	
&	位 AND	左到右
^	位异 OR	左到右
	位 OR	左到右
&&	逻辑 AND	左到右
	逻辑 OR	左到右
?:	条件	右到左
=	赋值	右到左
*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =	组合赋值	
,	逗号	左到右
最低级		

1.11 小结

由于当今的编程环境的复杂性, 程序员必须使用各种设计工具、技术以及技巧, 以编

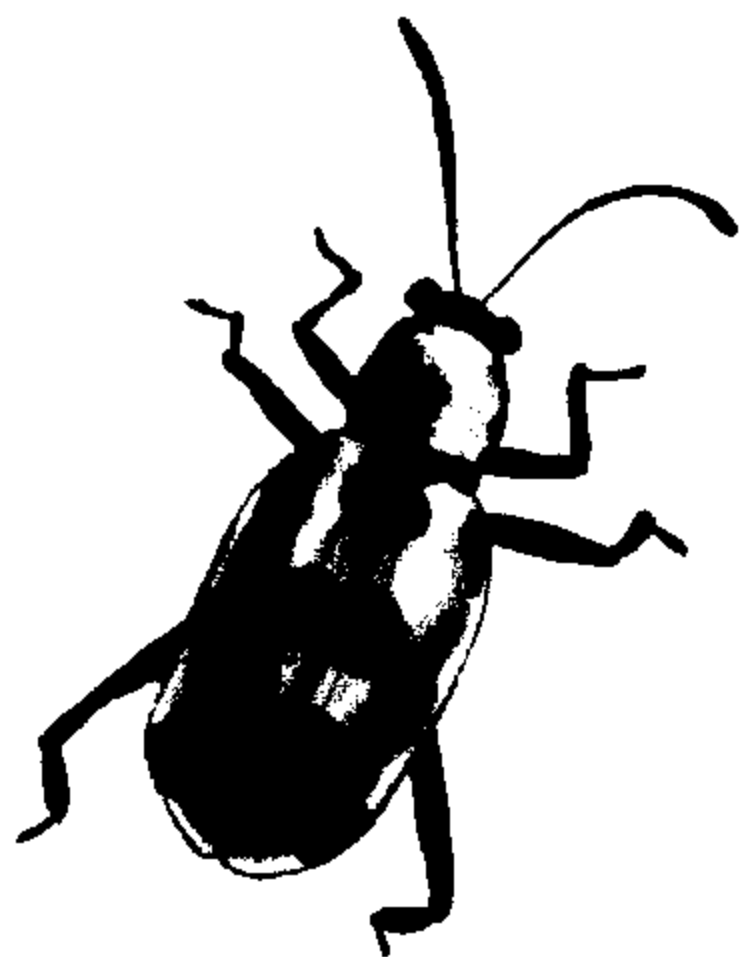
写出最可靠的算法，并且对于其他程序员来说要容易阅读和修改。这并不是一件容易的事情。

本章中，我们学习了应用程序开发者可以使用的所有设计工具，不论它是一个好的语言独立的设计，使用结构图进行“桌面检查”，谨慎的代码页间隔处理，还是匈牙利表示法，C/C++都提供了丰富的语法变化，允许我们增强源代码外观的简明的可视化。

在下一章中，我们将学习 C++ Debugger 的各种设置。这一章非常重要，因为我们可以 Debugger 设置阶段打开或关闭各种含义丰富的 Debugger 选项。在这一阶段中，稍不注意即可漏掉一些重要的 Debugger 选项，或产生混乱和无意义的应用程序专用显示格式。 ■

第 2 章

使用编译器优化





在上一章中，我们介绍了好的程序解决方案中语言无关的内容和那些特别与 C/C++ 语言有关的内容。本章中我们将进一步学习如何使用 Microsoft Visual C++ 编译器产生最有效的可执行代码。

许多程序员并不清楚在努力以最大的可能提高一个算法的性能时编译器可以提供的帮助有哪些，无论是从尽量少使用内存方面，还是从创建运行速度快的可执行代码方面。

所有的语言编译器、汇编程序及解释程序均完成同一件事情：将其各自的源文件语法向下翻译为只有微处理器可以理解的语言——机器代码。Microsoft C++ 的编译器也不例外。缺省情况下，Microsoft Visual Studio 产生一个所编写程序的调试版本。这是一种字面翻译，它在最终的可执行程序中产生一系列的低级机器指令，确切地表示了源文件中的高级指令。

相反，Microsoft Visual Studio 也可以产生一个发行版本(关于调试和发行的详细介绍见第 5 章)。这种选择使编译器具有了更大的灵活性，因为对源文件的逐字翻译并不一定是我们所希望的。建立发行版本的目标是产生一个尽可能小或尽可能快的目标代码，而不再为算法引入任何新的或不需要的行为。

本章首先介绍可以使用的编译器开关及其用法。然后，将演示如何通过重新编写少量代码与编译器开关一起，产生最整洁的可执行代码。

也就是说，本章讲述了关于代码大小与代码速度的问题。这两个因素并没有必要放在一起考虑。例如，一个航天工程师可能通知我们整个的航天飞机程序必须在 5K 的内存下运行。在这种状况下，最小化代码长度将是我们的目标，在代码设计中可能将采用一些比较自由的方法，从表面上看，可能如同某些黑客匆匆凑成的一个解决方案。另一方面，为了产生最快的执行速度，程序员可能打破其他一些设计原则，如大量使用全局变量，以最小化调用堆栈的使用。一般来说，与较简单和较直接的方法相比，较快的算法通常需要更多的代码。

2.1 编码的责任与编译器的优化

创建最快的可执行程序需要软件工程师对优化编译器的性能有一个准确的理解。

让我们快速查看我们的任务是什么。首先，请看下面的代码段：

```
for( int iOffset=0; iOffset < MAX_ELEMENTS; iOffset++)  
    iArrayOne[iOffset] = iOffset;  
for( int iOffset=0; iOffset < MAX_ELEMENTS; iOffset++)  
    iArrayTwo[iOffset] = iOffset;
```

虽然从代码的功效角度来看上述语法并不存在内在的错误，但该代码段可以重新编写为如下形式：

```
for( int iOffset=0; iOffset < MAX_ELEMENTS; iOffset++) {  
    iArrayOne[iOffset] = iOffset;
```



```
iArrayTwo[iOffset] = iOffset;
}
```

重新编写的代码将两个循环合并到了一个循环中，完成了逻辑上相同的任务，并且不再有第二个 **for** 循环的开销。从技术上讲，这种性质的代码重写称为阻塞，Microsoft Visual C++ 不能识别这些类型的代码优化，这是程序员实现时的事情。

然而，优化编译器可以针对机器进行一些技巧性处理，可以从此处节省一个字节或从另一处节省一个时钟循环，对整个程序将这些节省积累起来后将非常显著。例如，下面的语句：

```
and dword ptr mytable[si],0
```

从占用内存的角度讲，比下面的等价语句少占用三个字节，但其执行速度要比下面的语句慢三倍：

```
mov dword ptr mytable[si],0
```

不幸的是，由于微处理器每秒钟内要执行数百万条，并且很快即可达到数十亿条指令(或时钟滴答)，所以一个程序需要大量积累这些类型的时间节省技巧，以实现显著的性能转换。最好的设计方法是，首先考虑高效率的算法设计技术，然后再使用编译器优化解决性能问题。

在实际中，只有算法方面的优化才能产生执行速度的明显改进，低级别的优化通常并不能节省人可以感觉到的数百万个时钟周期。只有一个例外，即优化技巧应用于需要执行数百次的特殊循环或函数。Microsoft Windows 特别鼓励使用小的可执行代码，避免产生一个程序最大化共享内存的缺页中断。缺页中断将强迫 Windows 从磁盘重新加载内存，这是一种代价很高的操作。

2.2 Microsoft Visual C++的优化

表 2-1 中列出了最常用的 Microsoft Visual C++优化技术。其中将这些选项分解为两类：可能产生最小可执行代码的优化和可以提高执行速度的优化。

表 2-1 Microsoft Visual C++编译器的优化

优化技术	用于产生高效率可执行代码	用于产生最小化代码长度
调度指令	√	
函数级连接		√
字符串集中控制		√
register 键字的使用	√	√
常量和复件的传送	√	√
删除无用代码和无用存储器	√	√
删除冗余子表达式	√	√



(续表)

优化技术	用于产生高效率可执行代码	用于产生最小化代码长度
优化循环	✓	✓
强度降低	✓	✓
内联键字的使用	✓	✓
结构指针的省略	✓	✓
关闭堆栈检查	✓	✓
堆栈重叠	✓	✓
允许函数调用使用别名	✓	✓

下面的讨论将解释每一种 Microsoft Visual C++ 优化技术的使用。

2.2.1 调度指令

今天, 使用最新 Intel Pentium 超级处理器的程序员可以利用这些新的芯片所固有的硬件能力, 同时执行两条指令, 并且两条指令各自在其自己的管线上。当然, 此处假设一条指令的结果与当前正在执行的指令是无关的。不幸的是, 没有经验的程序员可能由于不注意而产生这种混乱, 产生一种称为管线延迟的情况。通过使用指令排序, 编译器将得到指示, 重新安排机器代码指令的顺序, 防止这种相关现象出现。

下面的例子说明了一个不可避免的管线延迟:

```
add      ax,shortInteger    ; 语句 1
mov      bx,ax              ; 语句 2
sub      dx,dx              ; 语句 3
```

如果微处理器同时执行语句 1 和语句 2, 则将产生不可预料的结果, 因为 ax 并不存在一个 shortInteger 拷贝。另一方面, 语句 3 的执行既独立于语句 1, 也独立于语句 2。简单地重新排列上面的三条语句, 即可消除该问题, 如下所示:

```
add      ax,shortInteger    ; 语句 1
sub      dx,dx              ; 语句 3
mov      bx,ax              ; 语句 2
```

虽然指令调度可以改进运行时的性能, 但其只对 Pentium 级的微处理器起作用, 并且语句的重新安排并不能影响代码的大小。

2.2.2 函数级连接

可能存在着这样的情况, 很好的代码设计与一组恰当的优化开关相结合, 优化一个函数(right out of existence)。这种情况对于内联函数或当编译器识别到其可以以函数本身从不被调用的方式编译该程序时是可能的。然而, 函数本身仍必须编译, 并包括到目标文件中。

不幸的是，编译器并没有识别其他模块是否访问该函数的方法，只有连接器可以识别何时一个函数从来不被激活。

函数级连接保证了一个源模块中的所有函数均以 Common Object File Format(COFF)，通过 COMDAT 记录编码为目标代码。这种信息允许连接器辨别非激活函数，以将其从可执行文件中删除。如果没有 COMDAT 记录，这一过程是不可能的。

2.2.3 字符串池

这一相对简单的优化指示编译器识别所有重复的字符串定义。当这种情况发生时，编译器只是简单地替代引用地址为该字符串的唯一版本。

2.2.4 使用 register 键字

使用 **register** 键字是可以同时减少代码长度并改进运行性能的首选优化技术。**register** 键字与局部变量类似，仅仅在内层有效——也就是说仅仅在一个函数内有效。原则上讲，键字指示编译器为相关变量指定寄存器存储器，而不是执行堆栈分配。

实际上，**register** 键字要求编译器为相关变量指定寄存器存储器。这一方法不仅节省堆栈空间，而且通过将变量保存在寄存器中保证了可以以最快的速度访问它，因为处理器读写自己的寄存器时要比读写内存快得多。由于寄存器编码局部变量，在代码长度方面也有一些减小。

Microsoft Visual C++编译器并不能识别 **register** 键字。相反，编译器自动审查每一个内部和外部变量的使用方法，巧妙地在寄存器与内存之间处理数据。几乎所有的数据对象都可以使用寄存器存储器，包括常量值、结构成员、函数参数以及由引用传递的参数指针。

这种决定是基于复杂的观察，由编译器确定数据的具体使用方式，将一个加权值指定给每一个与从存储变量到寄存器中所获得的好处相关的数据对象。当编译器产生目标代码后，在可能的情况下将最高分值的变量放到寄存器中。

最后给出一条技术说明：如果所使用的是其他 C++编译器，那么要注意应使用地址操作符**&**的函数语句书写寄存器变量，如下所示：

```
void somefunc( void )
{
    register int iregValue;
    int *piregValue;
    piregValue = &iregValue;
    ...
}
```

此处，我们真是如履薄冰，因为这需要 C++编译器厂商决定是对这种组合思想产生一



条警告或错误消息，还是仅简单地假设得到一个内存单元地址的需求要比寄存器存储器更为重要，并且缺省为标准内存地址分配。

说明

编译器并不认可用户的寄存器变量请求，相反当全局寄存器分配优化(/Oe 选项)打开时，编译器自己选择寄存器。然而，所有与 **register** 键字关联的其他语义都得到兑现。ANSI C 不允许使用寄存器对象的地址；这一限制并不适用于 C++。然而如果地址操作符(&)应用于一个对象，则编译器必须将该对象放置到一个其地址可以表示的位置——实际上这意味着要放置到内存而不是寄存器。

2.2.5 常量和复制的传播

从代码优化的观点来讲，访问保存在寄存器中的值比访问常量值更快，而访问常量比访问存储在内存中的值效率更高。如果一个代码段运行时不能再获得寄存器，则各种类型的存储方式将不再有区别，考虑使用常量替换表达式。这种方法允许编译器利用正向常量传播，在整个的代码段中重用该常量。编译器可以通过使用值本身替换计算常量值的表达式优化语句，如下所示：

```
iValueOne = 10;  
iResult = iValueOne;
```

通过常量传播，编译器对这两条语句按照如下所示的输入顺序编码：

```
iValueOne = 10;  
iResult = 10;
```

同样的方式，复制的传播也涉及到编译器对一系列的赋值中一个值正向从一个变量传递到另一个变量的识别问题，在这些赋值中的中间赋值并未使用计算式中的值，而是仅仅将其传送给下一个变量。在下面的语句中：

```
iValue = iSomeValue;  
iValue = myFunctionCall( iValue );
```

将 **iSomeValue** 赋予 **iValue** 实际上并未起到作用，优化编译器将识别这一顺序，将其重新编写并编码为如下的逻辑等价语句：

```
iValue = myFunctionCall( isomeValue );
```

此处编译器执行了一个无伤害参数替代，即由 **iSomeValue** 替换 **iValue**。由于执行了优化，所以第一条语句成了多余的语句，产生了在工业界称为死存储的现象。

2.2.6 消除死代码和死存储

在前面的一节中，我们学习了优化编译器如何跳过被识别为无意义的代码语句，并产生了一种称为死存储的情况。当然，了解到这些情况可能在前台发生，在编写程序语句时可以人为地加以避免，这只要在产生原始算法时小心一些即可。

所谓死代码有时也称之为不可到达的代码，指的是由于这样或那样的原因，按照算法的逻辑流程，确实不可到达、没有使用或“死掉”的那些代码块。如下的例子使用简单的 **if...else** 语句说明了这一概念：

```
bFlag = true;
if( bFlag )
    cout << "This statement always executes the true statement.";
else
    cout << "This is an unreachable statement.";
```

由于变量 **bFlag** 被初始化为了 **true**，这一 **if...else** 语句的 **else** 部分将永远不能到达。

然而，作为较早编译器优化的一种副产品，有时编译器对死代码或死存储并不产生目标指令，并自动在最终优化的可执行代码中消除这些情况。

2.2.7 删除冗余子表达式

与常量传播类似，当编译器检测到一系列的子表达式均访问同一个值时，编译器将只计算一次子表达式的中间结果，然后使用这一计算值替代所有的子表达式。例如，如下的两条语句：

```
iResultOne = iValueOne / iValueTwo;
iResultTwo = iValueOne / iValueTwo;
```

可以重新改写为如下形式：

```
iIntermediateResult = iValueOne / iValueTwo;
iResultOne = iIntermediateResult;
iResultTwo = iIntermediateResult;
```

虽然这一简单的重新编写消除了重复的除法操作，但如果这种中间计算非常复杂，那么这种优化的结果将非常显著。执行性能将与中间结果的重用次数成正比地得到提高。在这种情况下，简单的替换可以明显地减少代码长度并提高运行效率。

2.2.8 优化循环

当然，将前面讨论的优化技术结合到循环体中，将产生与循环的循环次数成正比的显



著效率。然而，还有其他的设计方法可以影响到循环的性能。请看如下的实例代码段：

```
for( int iOffset = 0; iOffset < MAX_ELEMENTS; iOffset++ )  
    iArray[iOffset] = iValueOne * iValueTwo;
```

循环体语句的固定部分为 iValueOne 和 iValueTwo 的乘法运算。通过如下方式的代码重写：

```
iTempResult = iValueOne * iValueTwo;  
for( int iOffset = 0; iOffset < MAX_ELEMENTS; iOffset++ )  
    iArray[iOffset] = iTempResult;
```

将乘法运算移出循环体外，减少了算法必须执行的(MAX_ELEMENTS * 表达式)次冗余计算。术语“提升”(hoisting)描述了将一个循环体移动到循环迭代语句之上的这一过程。

2.2.9 降低强度

降低强度或优化包括了编译器对复杂表达式的识别，这种复杂的表达式可以以一种更简单的形式重写，而不改变计算结果。大多数处理器加或减两个值时所使用的机器循环要少于乘或除。同样，有些乘法和除法运算可以使用更有效的位移操作符代替。例如，任何一个值被 2 的方幂乘或除，如下所示：

```
fResult *= 2;
```

上述语句可以使用更有效的左位移操作符代替，如下所示：

```
fResult << 1; //left by one bit effectively multiplies by two
```

这一简单的替换减少了大量的密集乘法操作时钟滴答，节省了 30 个以上的机器循环。值得注意的是，这种节省本身在运行时人们并觉察不到，但是正是这一个个优化的积累，产生了显著的性能提高。

2.2.10 inline 键字的使用

许多程序员并不注意每当一个函数调用被编译后所产生的大量机器代码语句，他们所看到的全部内容就是一条单个语句

```
myFunctionCall( arg1, arg2, arg3,...);
```

函数调用需要为调用过程结束时所要执行的下一条指令跟踪地址，需要将实际参数的复件压栈，需要更新指令指针使其指向第一条被调用子程序语句，需要编码弹出所传递参数，需要保存所有计算结果和返回结果的分配语句等等。当函数调用发生在循环体内时，对性能的影响非常明显，这要依后台的执行顺序而定。

说明

在 16 位 Windows 下，堆栈检查将激活一个调用，该调用是一个称为堆栈探测的 C 运行函数。在 32 位 Windows 下，当试图访问应用程序堆栈底部时，Windows 通过分配更多的堆栈内存，自己给出响应。

C++ 允许程序员通过声明一个函数为 `inline` (内联)，消除这种瓶颈。C++ 中的 `inline` 函数类似于宏，但由于它需要一个函数原型，所以效果要更好一些。这样将允许编译器在一个函数的实际参数与形式参数列表之间编译时检查。真正的 C++ 宏在使用不正确的值激活的情况下是满无目的的。

`inline` 函数不使用调用堆栈，不更新指令指针使其指向某些其他内存段的开始处，并且在正确使用的前提下，可以显著地改进运行性能。

2.2.11 省略帧指针

帧指针的优化仅仅在 Microsoft Visual C++ Professional (专业) 版和 Enterprise (企业) 版中有效。/Oy 选项禁止在调用堆栈中创建帧指针，这个选项提高了函数调用的速度，原因是不再需要建立和删除帧指针。对于 Intel 386 或更高的 CPU，它还节省一个寄存器 `EBP`，可以用于保存频繁使用的变量和子表达式。

使用 `EBP` 作为帧指针是为运行于 Intel 80286 处理器而设计的旧版本 Windows 的一个不必要的遗产。当帧指针被忽略时，编译器相对于 `ESP` 寄存器而不是 `EBP` 寄存器引用堆栈数据。一个函数的开始处变成了一条指令，该指令调整 `ESP` 堆栈指针创建堆栈帧。如果没有这一选项，编译器将为每一个需要堆栈帧的函数产生开始代码，将处理器的 `EBP` 寄存器指向帧的顶部。忽略帧指针的缺点是相对于 `ESP` 编码内存引用要比相对于 `EBP` 的引用多占用一个字节。

为在开发环境中找到这一选项，单击 Project 菜单中的 Settings。单击 C/C++ 标签，然后单击 Category 框中的 Optimizations，在 Optimizations 下选择 Customize。

2.2.12 关闭堆栈检查

在讨论关闭堆栈检查优化之前，有必要查看 Windows 如何添加内存到堆栈中。Windows 以页形式在堆栈上分配空间，页的大小与计算机的体系结构有关。对于 Intel 微处理器，每页分配 4K。分配这种附加空间的事件在某次存取落入堆栈的末端之外，进入一个称为保护页 (guard page) 的区域时发生。保护页是分配给堆栈的最后页，当应用程序到达该页时，Windows 提交另一个页以增加堆栈大小，这称为增加 (growing) 堆栈。

当程序所需要的堆栈空间大于可以从保护页上所获得的空间时，程序将试图渗透到保留内存中，如下所示：



```
void myFunction( void )
{
    char cArray[2 * 4096]; // allocates 2 pages or 8 Kb
    cArray[8000] = 'a';
    ...
}
```

问题并不会立即出现，因为编译器为自动数据分配空间是通过按照所要求的 8K 递减处理器的堆栈指针 **ESP** 执行的。然而，这并不会引起 Windows 分配更多的堆栈空间。如果为数组分配的空间开始接近于堆栈的底部，存取一个接近于该序列末端的元素可能使得堆栈的保护页到达保留内存区。此时，Windows 将使用非法存取错误结束应用程序，并立即引起用户注意。

当打开检查堆栈时，C++编译器将计算每一个函数的局部变量的总大小。当一个错误的定义产生一个可能延伸到保护页的对象时，编译器产生一个 C 运行时库的堆栈检查过程调用。这一 C 函数只是简单地以 4,096 字节的增量存取堆栈的连续页。

不幸的是，虽然检查堆栈可以建立一个更加可靠的应用程序，但其增加了一个程序的开销，并不必要地降低了运行效率。同样，另一种方法是通过小心地重新编写该子程序完成。在前面的例子中，触发存取错误的语句为：

```
cArray[8000] = 'a';
```

而不是：

```
char cArray[2 * 4096];
```

切记，只有在代码语句引起实际内存存取时，存取错误才会发生。然而如果子程序在 4,096 字节增量之内顺序访问数组元素，则该程序将正确执行，如下所示：

```
cArray[4096] = 'a';    // page one access
cArray[5100] = 'a';    // page two access
```

2.2.13 覆盖堆栈

为了使这一优化有效，应用程序必须使用短期的内部或局部变量声明。当关闭这一优化后，所有的内部变量都有一个唯一地址空间，而 Windows 的响应则是悄悄地增加其堆栈——这是一种很耗时的操作。当覆盖堆栈优化打开时，编译器将重用堆栈空间，用来保存那些活动未覆盖的内部变量。这将降低程序运行时堆栈超限的可能。

使用覆盖堆栈优化，程序可以获得双倍的性能提高，同时也最小化了堆栈中局部变量与堆栈帧顶部的距离。例如，当局部变量位于自帧指针 128 字节之内时，每一个变量的地址均将小于 3 个字节。

2.2.14 函数调用之间允许使用别名

有时当谈到覆盖定义、别名时，指的是一个算法使用多个名字访问同一个内存地址。这种类型的编码最常见的源代码要使用到 `union` 或指针变量，如下所示：

```
char cValue, *pcValue = &cValue;
```

此处的两个变量 `cValue` 和 `pcValue` 访问的是相同的字节。这种类型的代码风格可能将削弱某些类型的编译器优化——例如，由于一个代码语句需要一个变量的内存分配地址 (`pcValue = &cValue`)，从而不允许变量接收寄存器存储(`pcValue`)。

当程序员激活 `Assume No Aliasing` 优化开关时，实际上程序员正在指示编译器变量没有隐式关系或别名。这使得编译器直接主动地优化包含指针的代码，Microsoft Visual C++通过一个中间开关 `Assume Aliasing Across Function Calls` 使得这一优化开关更加出色。此处程序员为编译器许诺，除了函数的调用之外，代码中不再有别名存在，如下所示：

```
int iExternalArray[MAX_ELEMENTS];
void main( void )
{
    int *piExternalArrayOne = iExternalArray,
        *piExternalArrayTwo = myFunctionReturningArrayAddress();
}
int * myFunctionReturningArrayAddress( void )
{
    return( iExternalArray );
}
```

在这一例子中，别名存在于 `main()` 和 `myFunctionReturningArrayAddress()` 之间，这两个函数都需要 `iExternalArray` 的起始地址。当 `Assume Aliasing Across Function Calls` 激活时，编译器将允许优化所有包括指针的代码，正如上述例子所示。

```
char cValue, *pcValue = &cValue;
```

现在，编译器可以继续编译，并确定寄存器保存对于变量 `cValue` 和 `pcValue` 是否均有利。

2.2.15 全局优化

这一更大范围的选项捆绑了若干种优化技术，包括窥孔优化、登记(类似于 `register` 键字请求)、循环优化以及消除死代码和死存储。

2.2.16 产生内部函数的内联

表 2-2 列出了这些内部函数。当选择了 `Maximize Speed` 优化时，编译器将使用这些函



数的内部形式，有效地使用对应的内部内联形式替代每一个函数的调用。正如所有的内联替代一样，此处的替代也可以显著提高程序的速度，但存在着程序长度增加的可能。对于注重程序大小的应用程序，内部函数实际上比标准函数调用更为糟糕，以 `strcpy()` 为例，该函数的内部形式需要大约 41 个代码字节，而函数调用则只使用 18 个字节。

表 2-2 Microsoft Visual C++ 内部函数

<code>_disable</code>	<code>_enable</code>	<code>_inp</code>
<code>_inpw</code>	<code>_lrotl</code>	<code>_lrotr</code>
<code>_outp</code>	<code>_outpw</code>	<code>_rotl</code>
<code>_rotr</code>	<code>_strset</code>	<code>abs</code>
<code>fabs</code>	<code>labs</code>	<code>memcmp</code>
<code>memcpy</code>	<code>memset</code>	<code>strcat</code>
<code>strcmp</code>	<code>strcpy</code>	<code>strlen</code>

2.2.17 优化 math.h

虽然 `math.h` 中的函数原型没有真正的内部对应函数，但当选择了 `Generate intrinsic functions inline` 优化时，所有使用 `math.h` 的程序均将获得额外的执行速度的提高。虽然这些函数没有对应的内部形式，但编译器通过直接将函数参数放置到处理器中，而不是将其压入堆栈中，对其执行性能优化。当然，这将需要更多的编码，但获得了更快的执行速度。

2.3 Microsoft C++ 的优化开关

切记，只有 Visual C++ 专业版和企业版才支持代码优化。表 2-3 中描述了所有可以使用的开关。所有这些选项也可以通过各种 Visual Studio 菜单和对话框获得，这些内容将在本书中随时讨论。

表 2-3 Microsoft Visual C++ 优化开关

专业版/企业版的 优化开关	描述
<code>/G5</code>	对 Pentium 处理器优化代码
<code>/G6</code>	对 Pentium Pro 处理器优化代码
<code>/GB</code>	对 80386(/G3)、80486(/G4)、Pentium(/G5)及 Pentium Pro(/G6)选项的组合优化
<code>/G3</code>	对 80386 处理器优化代码
<code>/G4</code>	对 80486 处理器优化代码
<code>/Gd</code>	定义函数调用规范
<code>/Ge</code>	启用检查局部变量需要存储器函数调用的堆栈
<code>/Gf</code>	在可执行文件中相同字符串在一个位置的字符串假脱机
<code>/GF</code>	与/Gf 相同，只是字符串副本放置在只读内存中

(续表)

专业版/企业版的 优化开关	描述
/Gh	在每一个函数调用的开始处调用一个用户定义的函数, 可执行文件要更大和更慢一些
/Gi	启用增量编译, 即只编译那些自上次建立执行程序以来修改过的函数。可执行文件更大一些
/Gm	简化工作区中包含有许多自上次建立执行程序以来未修改过的文件的建立过程
/Gr	定义函数调用规范
/GR	向检查对象类型添加运行代码, 可执行文件更大
/Gs	启用堆栈检查
/GX	为自动对象在堆栈清退由 Windows NT 结构化异常或 C++ 异常触发时指定析构函数的调用时机
/Gy	启用编译器以 COMDAT 形式对单个函数的包装
/Gz	定义函数调用规范
/O1	最小化代码长度优化, 使用/Os 及其他开关
/O2	最大化代码执行速度优化, 使用/Ot 及其他开关
/Oa	用于不使用别名的源文件
/Ob	启用函数的内联扩充
/Od	关闭优化
/Og	消除局部和全局公共子表达式, 允许自动寄存器分配, 并允许执行循环优化
/Oi	使用内联函数扩充替代可优化的函数调用
/Op	保证浮点运算的精度
/Os	产生更小的可执行文件
/Ot	产生更快的可执行文件
/Ox	使用可能是最快的代码进行代码优化。Microsoft 推荐使用/O2
/Oy	关闭调用堆栈的帧指针创建
/Ow	提示编译器该应用程序不使用别名

2.4 使用 Microsoft Visual Studio 设置编译器选项

根据所建立目标的不同, Microsoft Visual Studio 预定义了一套其自己的缺省优化设置。例如, 当建立调试版本程序时, Developer Studio 关闭所有优化开关, 保证可执行文件是源文件的直接字面翻译。对于发行版本的建立, 缺省设置为以执行速度优化代码, 其代价是最终的代码长度可能将增加。这两种常用的建立目标程序的方法可以满足大多数的工程开发。然而, 有时可能需要手工调整一个工程的编译器优化选项。

首先选择 Project, 然后选择 Settings, 可以访问编译器的优化开关, 如图 2-1 所示。

实际的开关包含在 Project Settings 对话框中, 如图 2-2 所示。

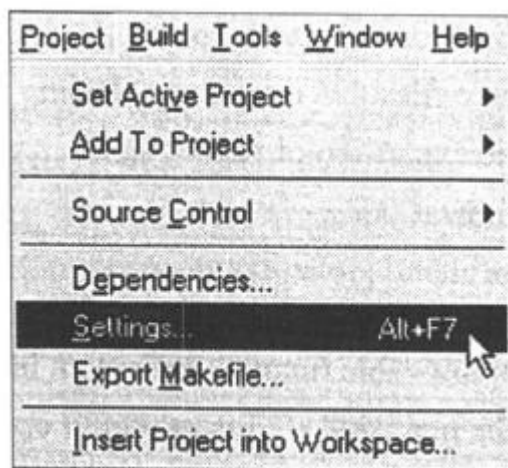


图 2-1 Project/Settings 选项

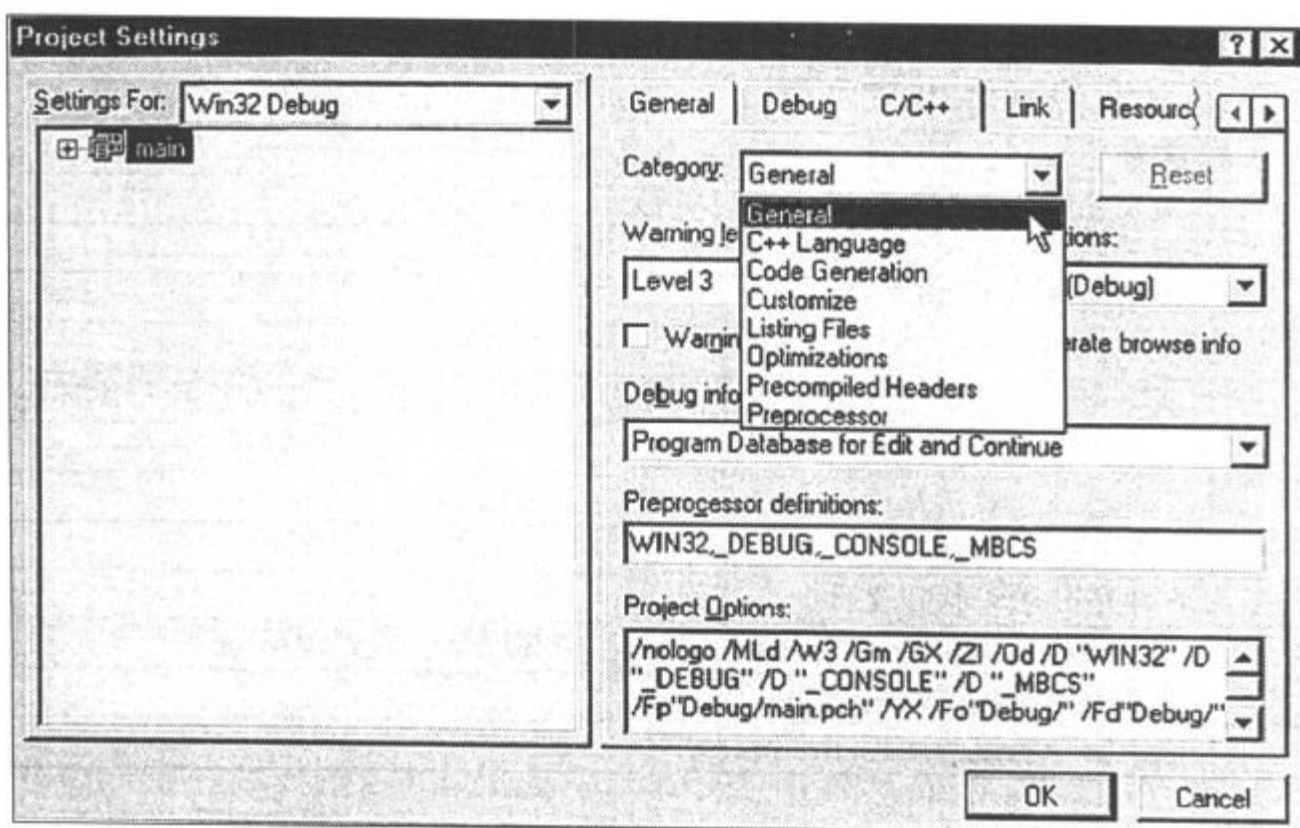


图 2-2 Project Settings 对话框

本节将集中介绍 Project Settings 对话框的 C/C++ 标签，其中包含了控制编译器优化一个工程的源文件的所有开关。

在设置任何优化开关之前，必须首先确定所要优化的文件。通过单击 Project Settings 对话框(图 2-2 中给出)，左侧的方格内部可以选择所要优化的文件。工程的源文件列表类似于 Workspace 窗口中的 FileView 窗格。在本例中，通过单击最高层索引 main 选择了该工程中的所有文件。

如果需要仅仅优化所选中的工程中的文件，则应该按住 CTRL 键后单击每一个所需的文件。这种方法允许微调文件优化，以满足特殊要求。工程中的一组文件可能需要优化速度，而另一组可能需要优化最小化使用空间。

初始的建立目标显示在 Project Settings 对话框的左上角，并依赖于该工程的当前活动配

置。当设置编译器优化时，目标应该为 Win32 Release。然而，应当注意的是该工程的活动配置并不改变，即使当选择其他设置时。

再来看图 2-2，优化选项列表依赖于 Category 下拉列表中的选择。对于编译器设置所列出的 8 种可能的类型中，有 4 种特别地包含了与编译器优化相关的所有开关。这 4 种类型是：

- **General**(图 2-2 中所选)

提供了最有效的选择一个一般优化目标的方法，但不允许精细地控制单个的优化技术。

- **Code Generation**

选择针对处理器的优化和一个工程的缺省调用规范。

- **Customize**

自动选择函数级的连接和字符串假脱机。

- **Optimizations**

允许精细调整一个工程的优化。

任何时候，当在 Category 下拉列表中选择和修改缺省设置后，均将激活暗淡显示的 Reset 按钮。这对于恢复下一个工程的全部初始设置很方便。

2.4.1 Project Settings 对话框中的 General 类型

当选择 General 类型(参见图 2-3)后，在 Optimizations 下拉列表中共有 5 种选择：Default、

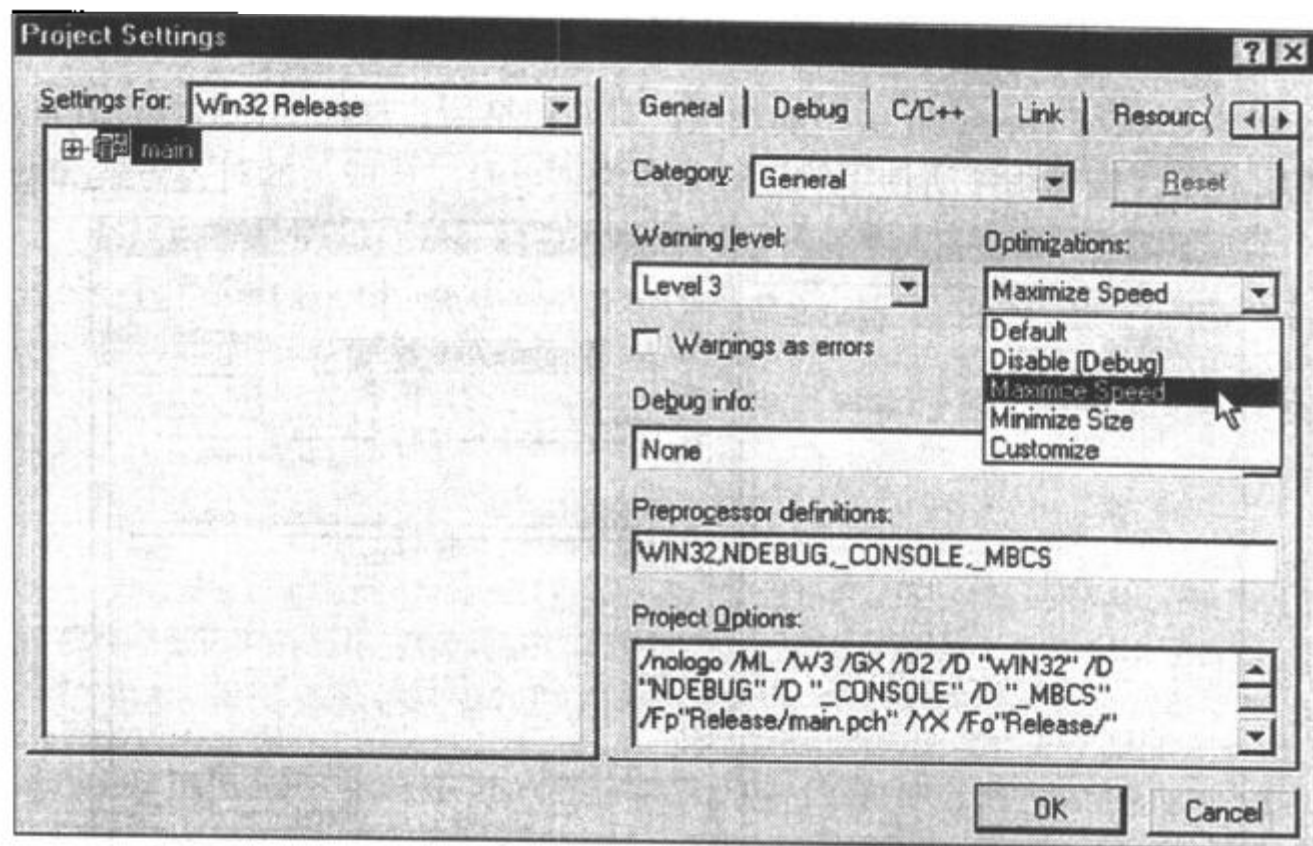


图 2-3 选择编译优化



Disable(Debug)、Maximize Speed、Minimize Size 和 Customize。Disable 将关闭所有的优化选项，产生一个逐字翻译源代码的可执行文件。当需要编译器执行一些所关切的更快速代码优化，而清除其他所有优化包括 Disable 开关时，可使用 Default 优化选项。对于函数级连接和字符串假脱机，Customize 优化选项提供了对优化开关的手工控制。

特别有趣的是选择 Maximize Speed 和 Minimum Size，这种选择最佳地组合了优化选项。表 2-4 详细描述了对 Maximize Speed 和 Minimum Size 起作用的优化选项。

表 2-4 General Category 的 Maximize Speed 和 Minimum Size Settings

优化选项	Maximize Speed	Minimum Size
产生内部函数内联	✓	无
快速代码优化	✓	无
最小代码优化	无	✓
全局优化启用	✓	✓
忽略帧指针	✓	✓
堆栈检查关闭	✓	✓
字符串假脱机启用	✓	✓
函数级连接启用	✓	✓

快速代码和最小代码优化指示编译器，当一个代码序列可以以两种方法之一优化时，优先选择的是哪一种，

2.4.2 Project Settings 对话框中的 Code Generation 类型

图 2-4 显示了 Code Generation 类型的选项，其中包括用户选择的针对特殊处理器的优化选项、缺省的调用规范、应用程序所使用的运行时库类型以及结构成员的对齐方式(alignment)。

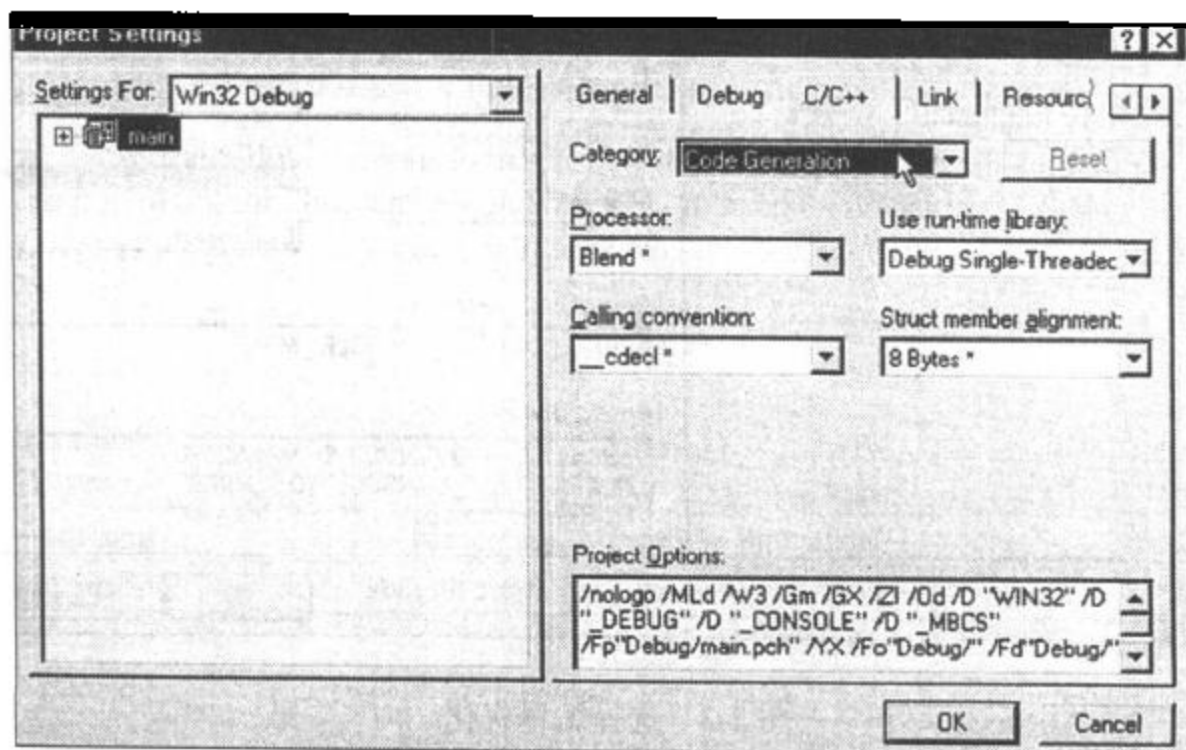


图 2-4 Code Generation 类型优化

2.4.2.1 选择处理器

Processor 选项指示编译器优化 80386、80486 或 Pentium 处理器代码生成。缺省设置 Blend 表示了一种折衷处理，这种处理侧重于 Pentium 组但增加了对前代处理器的选择优化。

2.4.2.2 选择调用规范

Visual C++ 允许三种调用规范：__cdecl、__fastcall 和 __stdcall(见表 2-5——注意该表为 C 代码或使用 extern “C” 键字的 C++ 算法定义了规范，C++ 使用的调用规范略微不同)。调用规范决定了自左至右或自右至左的参数传递、由何负责清除堆栈以及函数名字解释(mangling)(有时也称为签名或修饰)。

__fastcall 规范改进了 __cdecl 的缺省情况，但这只是在所调用 C 函数有至少一个参数时。当选择了该选项时，编译器将传送前两个参数到寄存器中(ECX 和 EDX)，而不是使用堆栈。所有其他参数则顺序地缺省传送到一般的 __cdecl 堆栈。

表 2-5 Visual C++ 可识别的调用规范

调用规范	传送顺序	堆栈清除执行者	解释风格
__cdecl	自右至左	调用程序	_functionName
__fastcall	自右至左	被调用程序	@functionName@nnn
__stdcall	自右至左	被调用程序	_functionName@nnn

nnn——表示参数列表中的字节数。

说明

__fastcall 调用规范不允许函数使用可变化的参数长度，并且当前的缺省关联寄存器为 ECX 和 EDX。Microsoft 不保证在未来的 Visual C++ 版本中有这种关系。

__stdcall 调用规范由 Windows API 使用，虽然这种规范从总体上减小了代码长度，但其也允许函数具有可变化参数列表。在这种情况下，调用是使用 __cdecl 调用规范实现的。

2.4.2.3 选择运行时库

正确选择合适的运行时库有助于减小应用程序的代码长度，虽然一般情况下缺省的运行时库是最佳选择。表 2-6 列出了 Use runtime library 列表中可以使用的选项。

表 2-6 Visual C++ 运行时库选项

库设置	使用	访问的运行时库
Single-Threaded(单线程)	静态连接到库，单线程	libc.lib
Multithreaded(多线程)	静态连接到库，多线程	libcmtd.lib
Multithreaded DLL(多线程 DLL)	为 Msvcrt.dll 加载库	msvcrt.lib
Debug Single-Threaded(调试单线程)	静态连接，单线程(调试版本)	libcd.lib
Debug Multithreaded(调试多线程)	静态连接，多线程(调试版本)	libcmtd.lib



静态或动态连接到一个运行时库所考虑的因素与静态或动态连接到 MFC 库相同。虽然静态连接使得可执行文件要大一些，动态连接使得代码要小一些，但动态连接要依赖于文件 Msvcrt.dll 的事先存在。

2.4.3 选择结构对齐方式

最后的设置定义了结构或联合成员的对齐边界。一般情况下，应用程序应该对所使用的数据类型和处理器，在很“自然”的地址处对齐结构的成员。例如，一个 4 字节的数据成员应该具有一个 4 的倍数的地址。

当可以选择小至 1 字节的对齐方式时，肯定将减少内存的使用量，但要注意：Pentium 级的处理器可以在一个时钟滴答中访问 4 字节的整数。在这些对齐边界内的访问需要一个具有三个附加周期的处理器延迟。积累起来将显著降低执行性能。

选择“自然”对齐边界在编写移植到多个处理器的代码时特别重要。一个没有对齐的 4 字节数据成员，即其地址不是 4 的倍数，在使用 80386 处理器时将引起性能损失，并且在使用 MIPS RISC 处理器时将产生一个硬件异常。在后一种情况下，虽然系统处理了这种异常，但其性能损失要更大。

2.4.4 Project Settings 对话框中的 Customize 类型

图 2-5 中给出了 Category 下拉列表中选择了 Customize 时的可用优化选项。可以看到，缺省时其中的有些选项是暗淡显示的，这是因为缺省情况下的优化设置为 Speed。为激活这两个暗淡显示的选项(Enable function-level linking 和 Eliminate duplicate strings)，需要选择 Minimum Size 或 Customize，这两个选项在 Category 列表选择为 General 时是可用的。

从前面的讨论中应该记住，缺省时函数级的连接仅仅适用于通过连接器使用 COMDAT 记录或 C++ 类内联成员函数标识的包装函数。启用 Enable function-level linking 将包装所有函数。

2.4.5 Project Settings 对话框中的 Optimizations 类型

Category 列表中的最后一个选项是 Optimizations。图 2-6 中给出了缺省的初始窗口。

Optimizations 下拉列表中可以选择的优化设置与 Category 列表中选择为 General 时的优化设置相同，其展开形式如图 2-7 所示，此处的选择为 Disable(Debug)。修改这一选择，则无论在 Optimizations 列表中(如图 2-7 中所示)，还是在 Category 列表选择为 General 时，均将导致其他 Category 下拉列表同步所选择的优化选项。

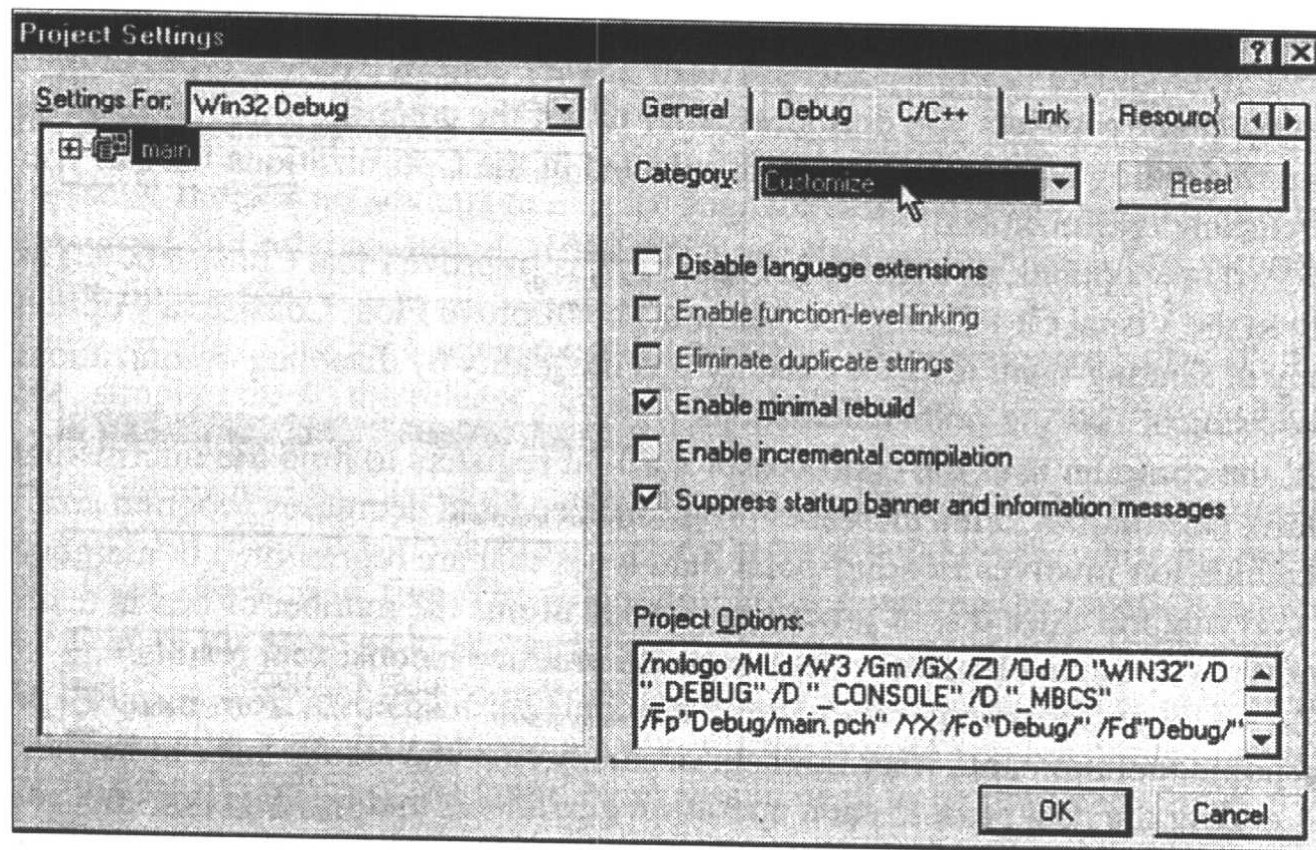


图 2-5 Customize 类型优化选项

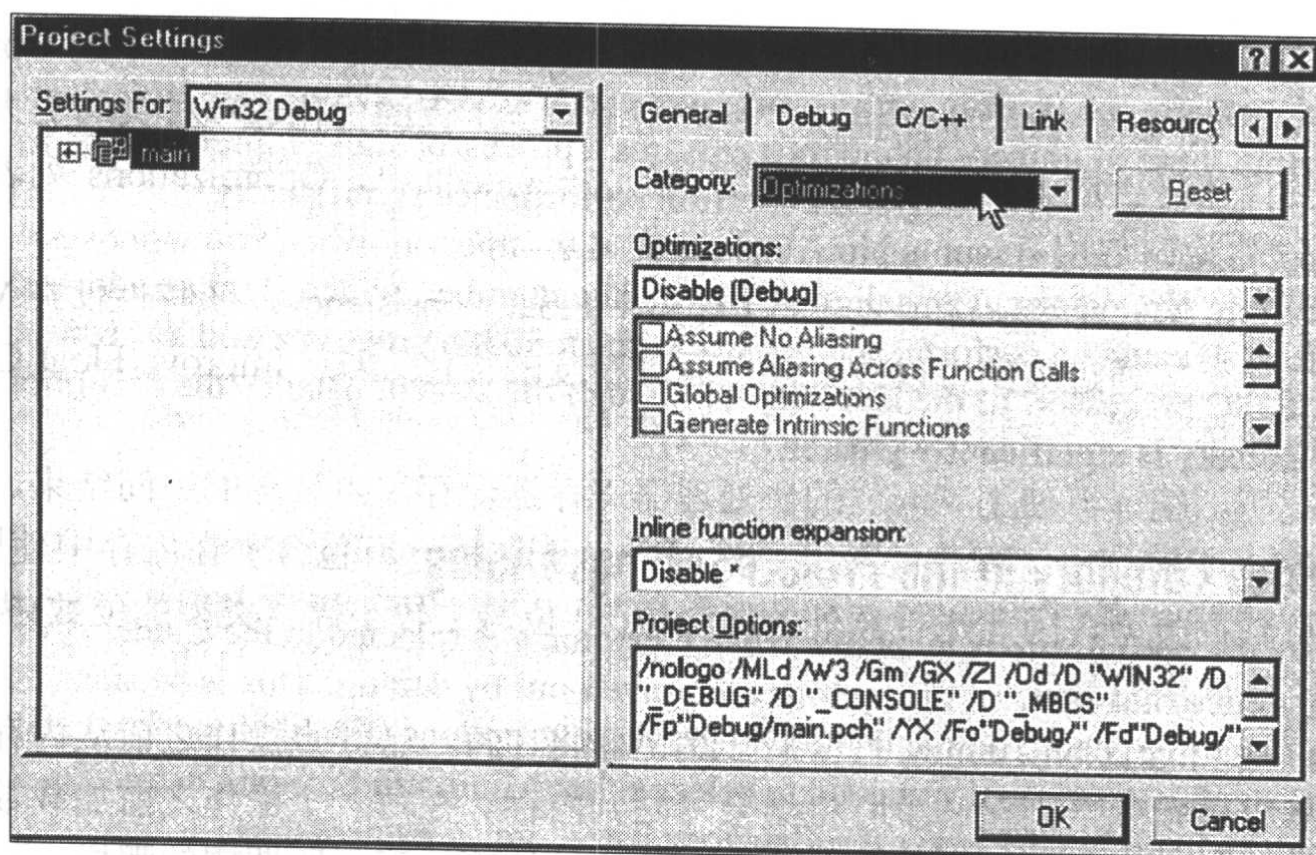


图 2-6 Optimizations 类型选项

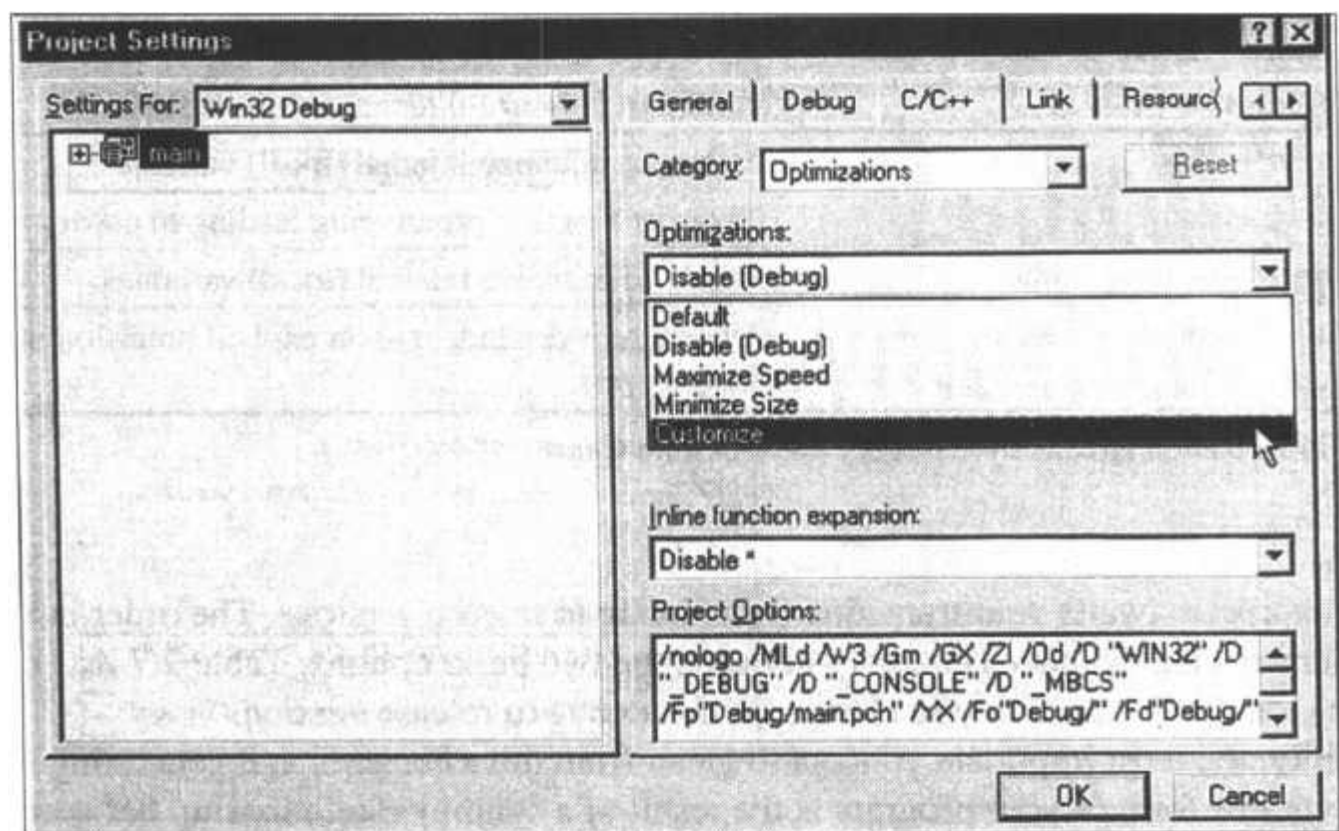


图 2-7 改变 Optimizations 列表中的模式

与 General Category 中的 Optimizations 不同, Optimizations Category 中的 Optimizations 的 Customize 选项(如图 2-7 所示)提供了对应用于一个工程的优化选项种类的更大控制,并且允许定义编译器是否应该扩展函数的内联。在这一对话框中, Optimizations 列表中选择了 Customize, 这将允许启用 Assume No Aliasing 优化选项。

图 2-7 中不可见的两个新选项之一是 Improve Float Consistency, 该特性仅在 Visual C++ 企业版中可以使用。通过关闭可以改变浮点运算精度的优化选项, Improve Float Consistency 选项改进了对相等与不相等的浮点测试的一致性。

缺省情况下, 编译器使用 80 位的协处理器寄存器保存浮点计算的中间结果, 这种方法提高了程序的速度并减小了程序的长度。然而, 由于这种计算包含了在内存中使用不到 80 位表示的浮点数据类型, 因此携带额外的精度位(80 位减去较小浮点类型的位数)执行过长的计算将产生不一致的结果。

当 Improve Float Consistency 打开后, 编译器将在每一个浮点操作之前从内存中加载数据, 并且如果赋值发生, 则在结束时将结果写回到内存。在每一个操作之前加载数据保证了该数据不会保留任何大于其数据类型容量的有效位。然而, 正如所期望的那样, 一个使用优化选项编译的程序可能比不使用优化选项时更慢和更大。

图 2-7 中不可见的第二个选项是 Full Optimization, 有必要对其进行一些解释。该选项为内联扩展和内部函数打开所有的优化选项, 产生快速代码、关闭堆栈检查并打开全局优化。

2.5 建立发行版本的建议

显而易见，在应用程序得到彻底调试和可靠测试之前，建立发行版本并没有什么好处。然而即使这样，在这一最后阶段仍存在一些令人惊奇的事情。

发行版本失败的一个相对简单的原因是存在着不可识别的别名，为定位这种烦人的代码，只要使用关闭 Assume No Aliasing 的方式重新建立该发行版本即可。如果建立成功，则定位该隐藏的别名。

在一定的环境下，关闭一个其局部变量需要一页以上的堆栈空间的函数的堆栈检查，可能引起发行版本的建立失败，而其调试版本在使用了堆栈检查时则运行很好。解决的方法是需要重新编写该函数，以触及每一个新的堆栈内存页或插入一个 check_stack 注记。

另外，在调试和发行模式下，对 C++ 的关键字 new 的编译也不同。在其调试版本中要增加额外的保护字节到内存中，而在发行版本中则省略这一代码。任何可能存在保护字节的代码段均需要重新编写。

从调试版本转换到发行版本的另一个惊奇的事情是函数参数求值的顺序在两个版本的建立中可能会有变化，表 2-7 中列出了从调试版本转换到发行版本时的优化选项和可能的原因。

表 2-7 优化失败的类型和可能的原因

优化选项	建立发行版本出错的原因
全局优化	初始化内部(局部)变量失败
内联扩展	初始化内部(局部)变量失败
帧指针忽略	不正确的函数原型导致了堆栈崩溃
产生内部函数内联	初始化内部(局部)变量失败
改进浮点一致性	在值比较中算法依赖于显式精度

毫无疑问，本章中所涉及的最重要的概念是，最有效的可执行程序形式的产生是代码设计与编译器优化选项共同作用的结果。

第 1 章“编写好的代码”和第 3 章“逻辑与语法错误”将有助于代码开发技能的提高，而本章及第 4 章“调试器基础”详细描述了可用于代码优化的 Visual Studio C++ 编译器工具。

2.6 小结

本章中，我们无疑已经学到了较以前更多的关于 Visual C++ 的 Debugger 功能的知识！一条好的消息是我们无须在任何时候都使用所有这些选项；一条不好的消息是如果不知道有

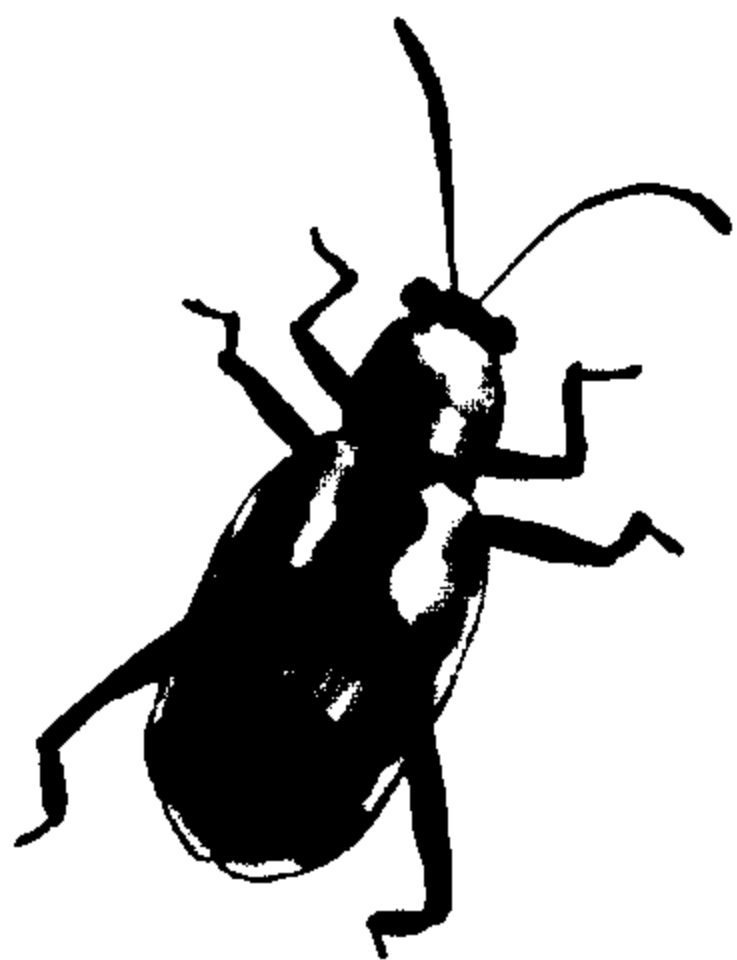


哪些 Debugger 选项可以使用或这些设置/开关所放置的位置，则有可能削弱 Debugger 有效检测和诊断代码错误的能力。

在下一章中，我们将学习算法可能产生的程序错误的类型，并致力于给出在设计阶段可以改变的预先警告，以防止编译和运行阶段的程序错误。 ■

第 3 章

逻辑与语法错误





对于初学编程的人来说往往很难意识到学习可以产生易读的设计和可靠算法的好的程序编程习惯的重要性，其原因是在学院环境下编写的程序常常与在实际情况下所开发的程序有很大的区别，实际中程序的风格与形式非常重要。学生的程序通常非常地小(通常不超过数百行代码)。这些程序只执行和修改少数的几次(绝大多数在交上后再也不执行)，几乎很少得到除这个学生和其导师以外的其他人的仔细检查，并且也不是在其预算的限制内开发的。

另一方面，现实世界中的应用程序可能很大(数千行的代码)，由程序员组所开发，通常要使用很长的一段时间，因此需要维护，使其保持最新和正确，并且这种维护常常是由初始程序员之外的另一个人完成。

在当今的编程环境中，硬件开销在持续降低，而程序员的费用则在稳步增长。与以往任何时候比，降低编程和维护开销的重要性，编写易于他人阅读和理解的程序的重要性，均显得更加突出，并在继续增长。当然，任何程序最重要的特征是其是正确的。

无论一个程序的结构设计是如何合理，也无论文档如何完备，或者程序看起来是如何地好，如果不能产生正确的结果，则其一文不值。但是众所周知，程序执行没有产生任何错误消息的事实，并不能保证其是正确的，由于计算机系统无法检测的逻辑错误的原因，因此所产生的结构可能是错误的。

检测和纠正错误是软件开发的重要组成部分，称之为确认与验证(validation and verification)。确认指的是检查算法和程序是否达到了问题的技术要求，验证指的是检查算法和程序是否正确和完整。有时将确认描述为回答这样的问题：该程序是否解决了所要解决的问题？而验证则是回答这样的问题：该程序是否正确解决了该问题？图 3-1 表示了一个优秀的程序开发策略。

本章中我们将逐一检查开发软件系统时可能出现的四种类型的错误。我们还将学习好的程序开发、设计思想，并重温防止错误的策略。本章结束时解释了如何使用 Visual C++ 的在线帮助工具，以跟踪警告或特定错误调试消息。

24x7

Visual C++ Debugger 自动以正在执行的语句的范围同步观察窗口变量。为继续观察一个变量的内容，从自动跟踪观察窗口(屏幕的左侧)单击后拖动该变量到窗口的右半部即可。

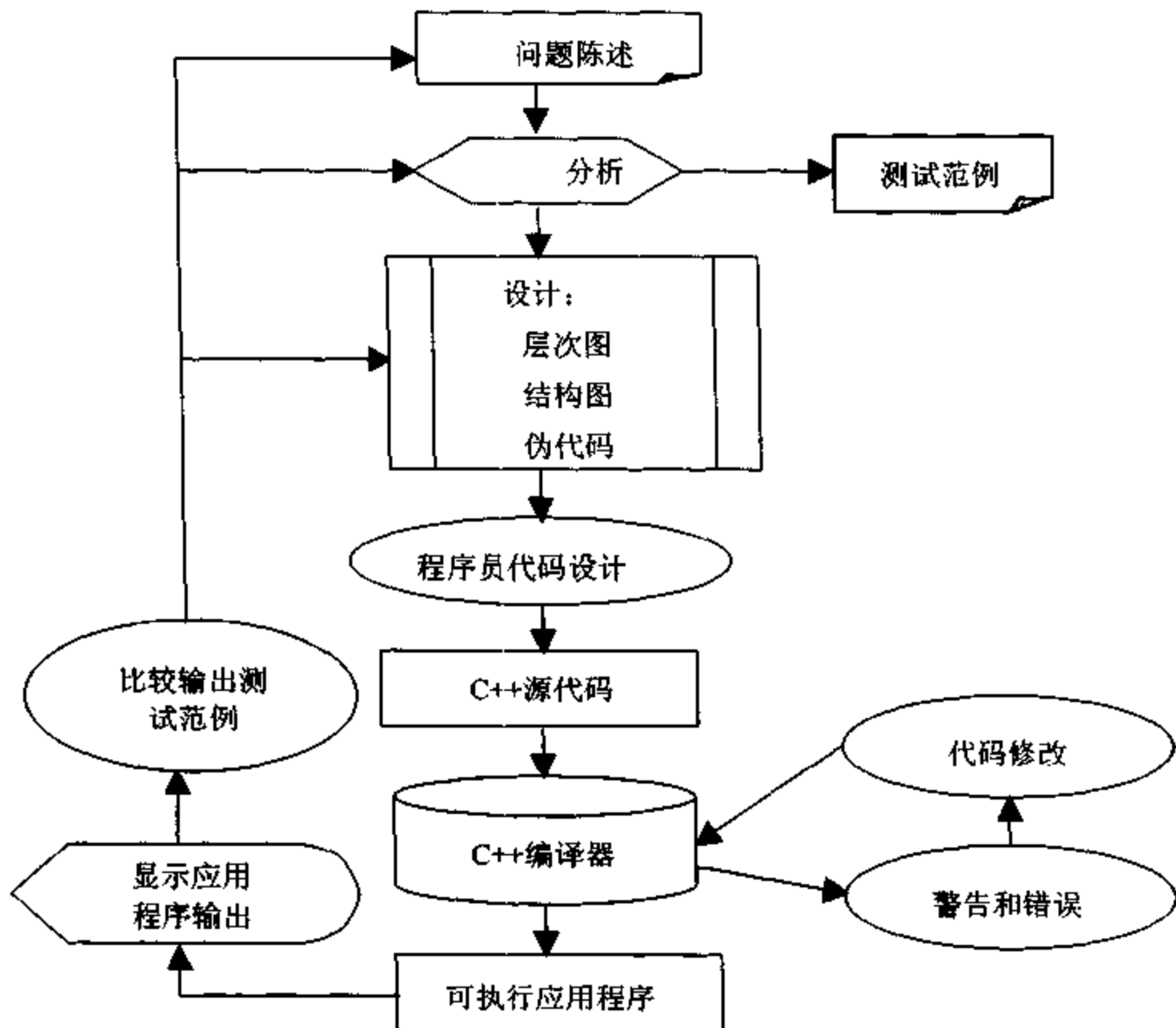


图 3-1 好的程序开发策略

3.1 好的调试策略

有经验的程序员早就认识到，使用一个系统和简单的方法调试一个程序的必要性。他们并不修改什么，仅仅因为其盼望程序将可以工作，并且不知道做其他什么事情。相反，他们利用各种技巧使用其资源孤立和纠正所存在的问题。这些资源包括算法、程序清单、集成 Visual C++ Debugger、参考手册、本书以及丰富的在线帮助文件。

逻辑和运行时错误通常是程序中一个严重缺陷的后果。通过盲目的修改程序是不能驱逐和纠正这种错误的。定位错误的一种好的方法是使程序打印引导性结果，以及指出什么时候该程序的某一特殊部分开始执行和结束执行的消息。



3.2 四种程序错误类型

应用程序可能遇到的基本错误类型有四种：

1. 语法或编译时错误

在编译期间出现的错误。

2. 连接错误

在用于建立可执行文件的连接处理过程中发生的错误。

3. 运行错误

程序运行时发生的错误。

4. 逻辑或意图错误

应用程序运行时并无错误，但产生的结构不正确。

这些错误中的大多数均发生在将所编写的 C++ 程序向其可执行形式转化的过程中。另外，正如在本书中将要看到的那样，根据所要建立的版本可执行文件的版本的不同(调试版本和发行版本)，可能出现不同的错误组合。

图 3-2 表示了编译和连接两个阶段。首先，编译器将源代码翻译为目标代码格式(目标文件格式几乎就是一个可执行文件，其中仅仅忽略了少数几个引用)。对于典型的 Windows 应用程序，需要将多个目标代码连接到一起。当然，所有这些都是由工作空间或工程文件所决定的。

3.2.1 语法错误

不论是对新手还是对有经验的程序员，最常见的错误类型都是语法错误，即程序中包含了违反 C/C++ 语法或语法规则之一的代码语句。在编译程序时将自动检测出这种类型的错误。

例如，错误地输入了如下所示的输出语句：

```
cout << "String-literal output  
      << "that exceeds one line";
```

其中包含了如下所示的有关致命错误：

```
-----Configuration: syntax - Win32 Debug-----  
Compiling...  
syntax.cpp  
c:\debuggingbook\chp03\syntax.cpp(7) : error C2001: newline in constant  
Error executing cl.exe.  
syntax.obj - 1 error(s), 0 warning(s)
```

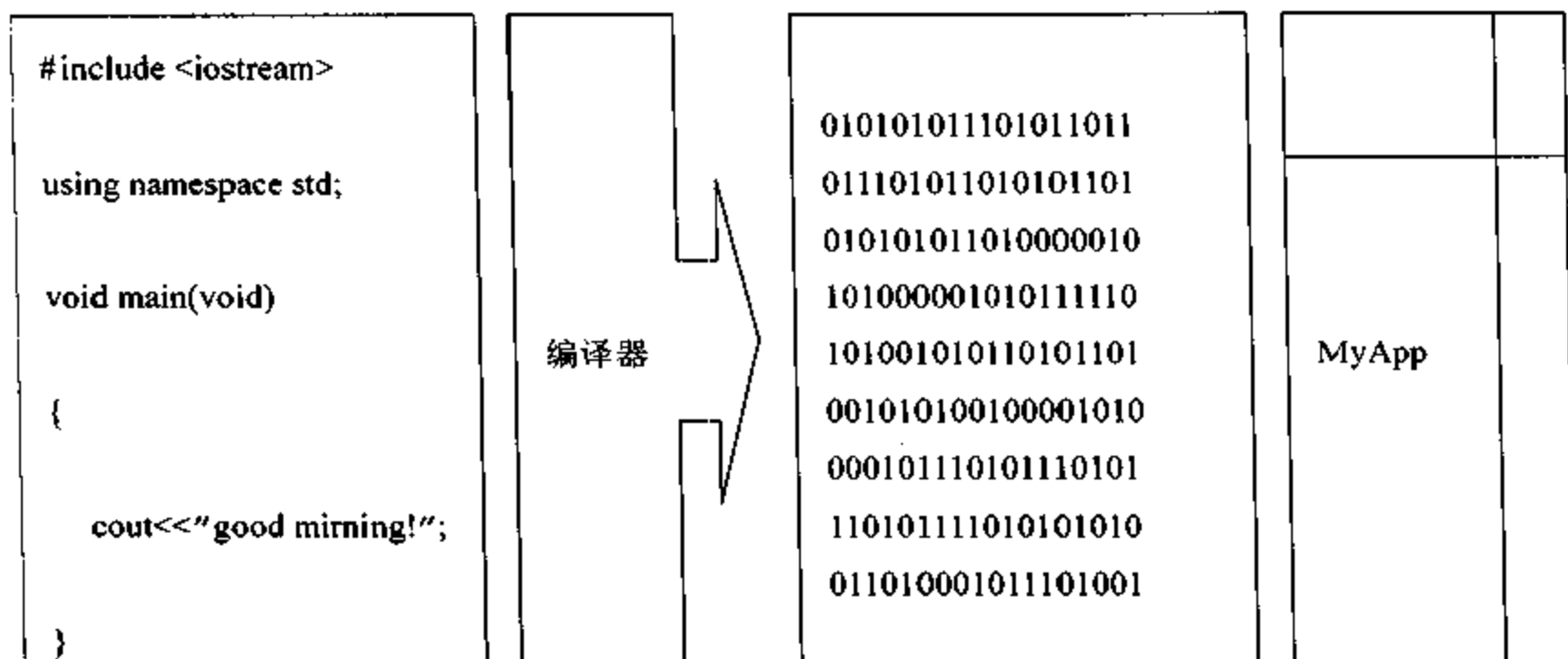



图 3-2 C++应用程序的翻译过程

初学编程者或冒失的程序员都容易犯这种类型的错误。当然，要改正这一错误只要使用双引号封闭第一行的字符串文字即可：

```
cout << "String-literal output"
      << "that exceeds one line";
```

虽然大多数程序员，包括初学者或有经验的程序员，均被语法错误所困扰，但语法错误实际上是最容易处理的错误。纠正语法错误非常简单，只要找出程序中违反 C/C++语法规则的位置，并纠正错误即可。随着经验的积累，程序员对编程语言语法规则的将更加熟悉，出现这种类型的错误的频率将降低。即使当语法错误出现时，也可更加容易地得到纠正。

当 C/C++编译器将源代码翻译为可以在计算机上运行的可执行程序时，编译器将试图定位和报告出尽可能多的语法错误，并对合法但以后可能引起错误的潜在问题给出警告。Visual C++编译器可以报告出声明但从未使用过的变量和在初始化之前对变量的使用，甚至可以指出所产生的逻辑流程将禁止某些代码段被执行——这种问题被标记为“unreachable code”（不可到达的代码）。

3.2.2 连接错误

出现程序错误的另一个源是当将编译过的程序与该程序所使用的库连接时。一个常见的例子是 Unix 系统中使用平方根函数 `sqrt()` 的程序，该函数的定义保存在数学库中。这一函数的声明在文件 `<math.h>`（对非标准应用程序）或 `<cmath>`（对使用命名空间的标准应用程序）中，并且要正确编译该程序需要使用 `#include <[math.h|cmath]>` 预处理语句。

然而这样的程序仍不能正确连接，因为标准的 Unix 连接器并不自动搜索该数学库，必



须使用 `-lm` 开关指示连接器实现这一目的，纠正连接错误常需要更多关于计算机系统的知识。

大多数情况下，当源代码试图与数百个 Microsoft 所提供的对象定义连接时，编译器将只是使用 `mIsScAsing` 作为一个给定标识符的名字，或产生一条如下所示的连接错误消息：

```
-----Configuration: syntax - Win32 Debug-----  
Compiling...  
syntax.cpp  
Linking...  
LIBCD.lib(crt0.obj) : error LNK2001: unresolved external symbol _main  
Debug/syntax.exe : fatal error LNK1120: 1 unresolved externals  
Error executing link.exe.  
syntax.exe - 2 error(s), 0 warning(s)
```

此时的问题是忽略了函数 `main()`！注意调试消息中的标识符 `main`，其前面使用了一个下划线 `_`，变成了 `_main`。这一下划线实际上指出了错误发生的时间。仔细检查前面的调试消息，可以看出该程序实际上已经通过了编译，只是为通过连接。

初学编程者看到这一类型的错误消息时，典型的反应是“奇怪，我已经查遍了所有的代码，我并没有使用标识符 `_main` 呀！”他们不能理解的是，有经验的程序员知道这是连接器正在报告其不能找到对一个损坏的函数名(有时称之为签名)和一个相应的已定义库程序的匹配。

当然，在这一例子中，调试诊断出的问题即忽略了 `main()` 的问题并没有什么新奇之处。在一个具有数千行源代码和数百个用户定义标识符的程序中，再加上所使用的编译器定义的宏和标签，这种类型的连接器消息一开始即可将任何程序员置于艰难的境地。

3.2.3 运行错误

当清除了所有的编译错误并使用连接器创建了可执行程序后，即可以开始程序的执行。但是在程序执行时错误仍然可能发生。这种运行时的错误可能由于一个计算机不能处理的事件的发生而引起该程序不正常结束(在其应该结束之前)。

当然，即使是已经没有语法和连接错误，对于一个执行被 0 除的不正确算术表达式、对于负数求平方根运算或对于试图输入无效数字数据等这样的错误也是无法捕获的。只有在程序开始执行后运行错误才能检测到，这也正是这种类型的错误称为运行错误的原因。例如，如下的程序在用户为 `iNumberOfStudents` 输入一个 0 时，产生一个运行被 0 除异常，而其他情况下都能正常运行：

```
#include <iostream>  
using namespace std;  
void main ( void )
```

```
{
    int    iNumberOfStudents;
    float  fClassTestTotal;
    cout >>
        "Enter the class test total: ";
    cin >> iNumberOfStudents;
    cout << "The Test Average is: "
        << (fClassTestTotal / iNumberOfStudents) << endl;
}
```

运行错误包括:

1. 硬件检测的错误, 如被 0 除、算法溢出、内存错误及设备错误。
2. 系统错误, 如文件操作失败或消息队列满。
3. 逻辑错误, 如越界数组下标或从空队列中删除元素。
4. 异常或应用程序特定错误, 如无效的输入格式。

在执行过程中异常将经常发生, 但一个可靠的系统必须做好处理这些异常的准备。最糟糕的响应是继续执行而处理, 这样所产生的结果将是无效的。然而, 在这样的情况下立即终止应用程序将增加语言实现的复杂性, 对于可能需要释放系统资源或恢复文件状态的生产系统这也不是一个正确的响应。

另外, 应该提供的是错误的一些自然迹象。在许多情况下, 如应用程序特定的异常发生时, 限制故障的影响并继续执行还是可能的——也就是说, 程序可以从错误中得到恢复并继续执行。

使用一个符号调试器查找这样的错误是最容易的。Visual C++ Debugger 允许通过依次执行程序的一个语句进行跟踪, 直至遇到产生错误的语句。一旦识别出错误, 则必须通过使用以正确语句替换错误语句的方法纠正这些错误, 并且修改后的程序必须重新编译、重新连接并再次执行。

常常有这样的情况, 检测到异常的子程序并不是需要处理的子程序。例如, 如果一个堆栈函数由于调用者弹出了一个空堆栈而发出堆栈溢出的信号, 则由该对象的使用者而不是堆栈模块的设计者指定将要采取的动作要更为恰当。同样, 检测到无效输入的模块可能并不是与用户通信的同一个模块。另外, 可以检测到错误条件的子程序在该系统中可能有许多其他的子程序调用它(正如堆栈弹出函数), 也可能是独立编写和编译的模块。

幸运的是 C/C++ 的许多操作均返回产生错误的错误码。每一个对这种操作的引用变成了一个条件语句, 这种条件语句测试潜在的测试条件并指定发生错误时所采取的动作。为了传送错误信息, 每一个子程序必须将这一信息传送给其调用者, 直到那个可以处理这一错误条件的子程序为止。

然而必须注意, 这种方法将这些子程序紧密联系在了一起, 如果子程序必须检测、传送或处理多个异常将很不方便。这种方法还扰乱了所有相关子程序的逻辑流程, 特别是错误



信息和其传送与这些子程序的目的不相关时。另外，可能发出一个错误信号的函数的调用者并不一定检查错误条件。

3.2.4 逻辑错误

在四种类型的错误中，逻辑错误是最难查找的一种，因为这种错误来自于对问题的解决方案的错误理解。由于编译器对于逻辑错误不能产生错误消息，这使得情况更糟。这种错误仅仅产生错误的结果，甚至可能中断应用程序。这样就需要使用各种各样的测试数据集测试程序的执行。只有通过以广泛的数据值彻底测试一个程序，才能确信该程序不包含逻辑错误。

另外，定位一个逻辑错误最简单的方法是，使用符号调试器跟踪每一条程序语句的执行。调试器允许在执行过程中的任何位置显示某一变量或表达式的值，可以比较正确的值与计算得到的值，当发现一个错误时，则定位出逻辑错误所在的源代码。

另一个处理逻辑错误的方法是，在程序中的各关键点上插入输出语句，显示不正确结果的计算中所涉及到的各个变量的值。通过手工执行跟踪，当输出值与预测值不一致时，可以在这一位置找到错误的根源。

定位逻辑错误要有足够的耐心。通过软件生命周期的前三个阶段中的细心努力，可以很容易地避免逻辑错误的发生，这三个阶段是问题分析和制定技术指标阶段、设计阶段以及编码阶段。

即使当程序已经完成并在开发者认为运行正常的情况下交付给了客户，仍可能有错误存在。有许多这样的软件实例，它们并未完成需要其完成的工作。这种情况是没有满足问题的技术指标。

当程序的需求方错误地描述问题时将发生有关的错误。这种情况可能是需求方不能肯定其需求到底是什么。技术指标中可能忽略了一些细微或关键的东西，需求可能没有描述清楚，或者——正如常常出现的情况一样——在程序开发已经开始之后需求方改变了其想法。

错误监视

如果考虑将 C/C++ 的一条语句等同于一个英语句子，那么数据对象就是一个句子中的名词，数据处理语句就是动词。当试图调试一个应用程序时，要在一个问题语句中寻找其名词和动词：它们通常提供了所需的输出、输入及处理的线索。名词指的是输出和输入，而动词则是处理步骤。

3.3 查看错误消息

当在 Visual C++ 中编译和连接程序时，建立执行程序的过程自动打开 Output 窗口，显

示关于这一建立是否成功的详细信息，包括所有的警告和错误消息。Output 窗口有两种方法可以帮助我们找到并修复产生错误和警告消息的原因，即通过自动跟踪产生消息的代码行和热连接到上下文相关的帮助主题提供关于该消息的更多信息。

为查看产生特定诊断消息的代码语句，只要双击 Output 窗口中相关的诊断消息即可。此时，相应的源代码文件打开，并显示一个指向产生诊断消息的行的指针。

为获得一条错误消息的帮助，首先在 Output 窗口中使用鼠标指针单击错误消息码(例如 C2001，参见图 3-3)，然后按 F1 键，则与所选择的号码对应的主题将在 Help 窗口中打开。

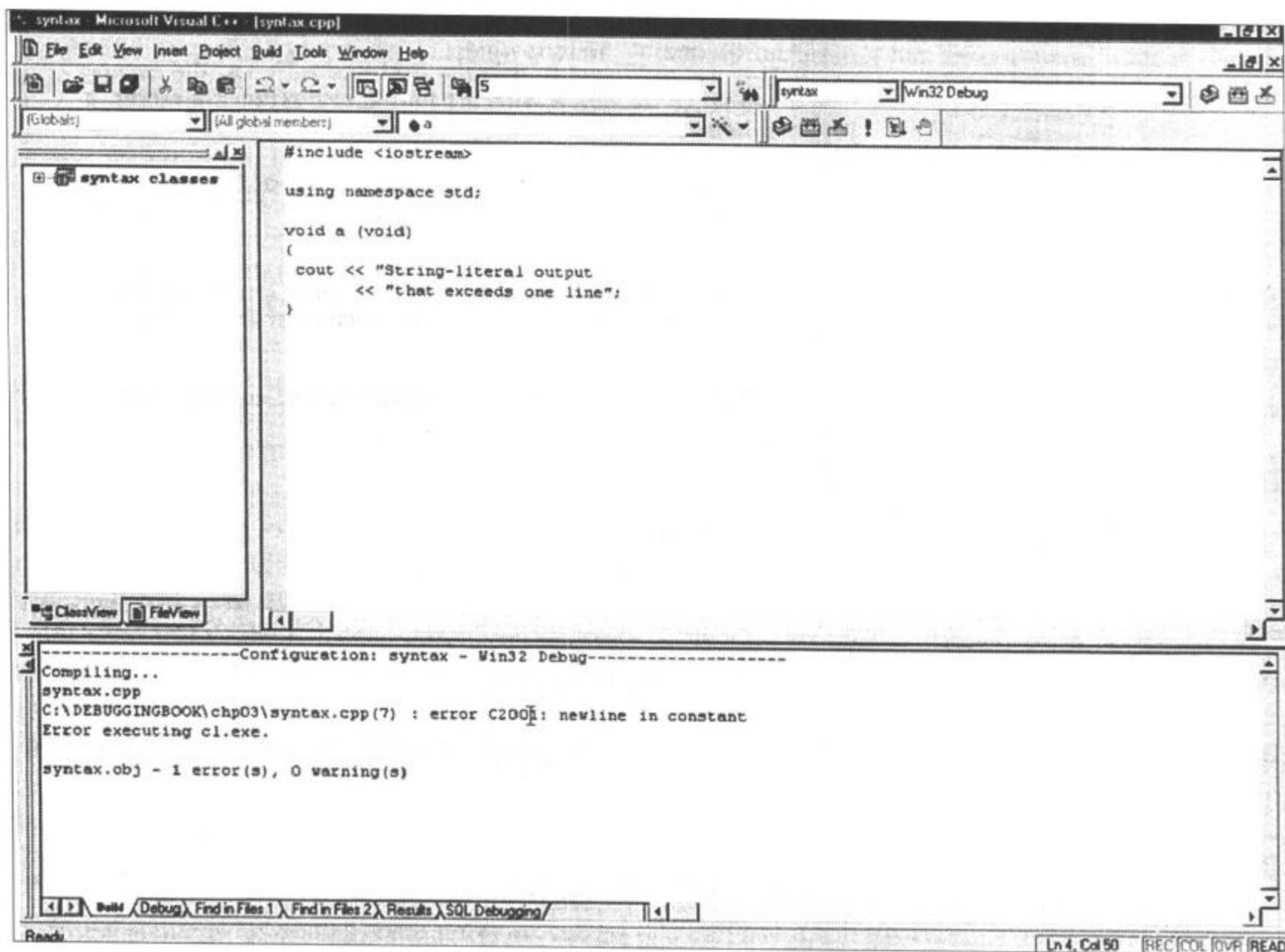


图 3-3 将 Debugger 设置为自动跟踪错误号码(C2001)

图 3-3 演示了获得错误 C2001 帮助的情况。图 3-4 中给出了相应的帮助文件。一般情况下，这些帮助文件将解释所遗漏的语法或逻辑语句错误，使得程序员可以修复这些语句。

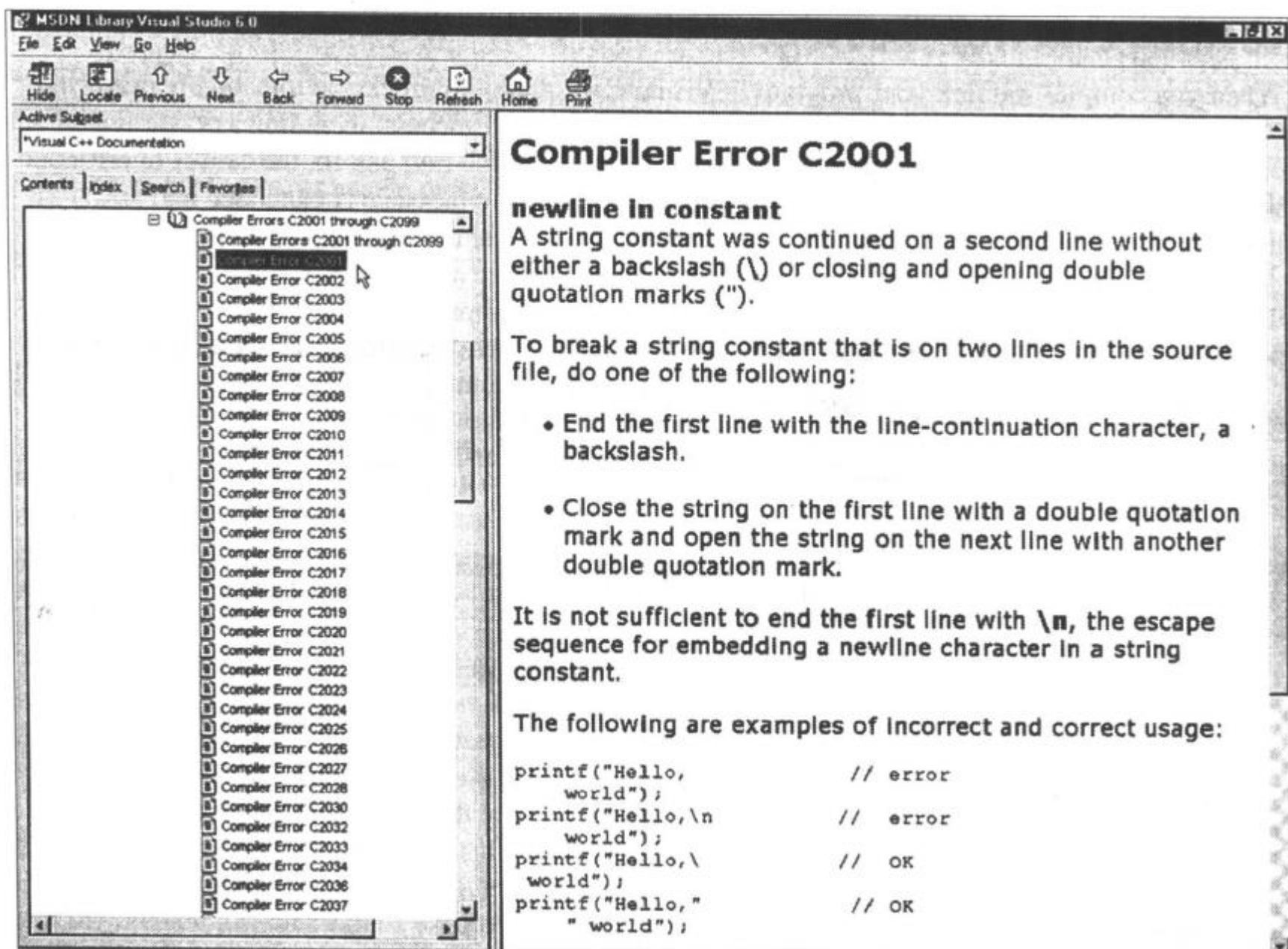


图 3-4 选择 C2001 错误后按 F1 的结果

正如 Visual C++ 所包含的帮助文件一样，这些帮助文件中常常包含有样例代码语句，演示错误究竟是如何产生的。

3.4 预防性维护

大多数软件工程师都知道，第一次实际运行所编写的程序而不出现任何错误，这种情况既罕见又令人欣慰。当然，一个好的问题定义和规划可以避免许多程序错误或程序故障。然而，总有一些程序错误是检测不到的，而无论所做的规划有多少。消除程序错误常常是整个编程过程中最耗时的工作。根据业界统计显示，一个程序员 50% 以上的时间要花在程序调试上。

3.4.1 桌面检查的含义

对于具有多年编程经验的人来说，可能回想起为价值百万美元的大型计算机编写并调试程序的时刻。这些庞然大物一般来说执行众多社团的重要任务，所以一个程序员一天只允许编译/运行一段时间。

如果读者现在还深深地记得那段“美好”的时光，一定为今天的开发环境而感激不尽。然而，如果因为从来不敢想象这样的恶梦，而使一天只能有一次编译/运行机会的追忆已远离而去，则需要停下来认真地回忆一下那时的情景。为什么呢？这是因为正是那种古老陈旧的编译/运行限制，造就了当今大批优秀的程序员。

如果一个软件工程师的目标是按时生产正确、有效、模块化并且可靠的算法，那么方法只有一种。这种方法要使用到一门好的语言无关的设计(Language-Independent Design)课程中所讲授的所有设计原则。当一个程序员一天只能有一次上机机会时，在没有确切相信算法可以通过的情况下，程序员是不会试图编译/运行的。程序员必须桌面检查并逐行查看其代码，否则即可丢掉饭碗。

至今仍然有效的一条经验法则是桌面检查一个算法。桌面检查一个程序类似于校对一封信或手稿，其思想是在内心对程序进行跟踪，确保程序逻辑正确。程序员必须考虑各种可能的输入，并记录程序执行期间所产生的所有结果。更特别的是，必须通过考虑输入不可能的非正常数据，确定程序的走向。在桌面检查一个程序时，必须永远牢记墨菲法则(Murphy's Law)：如果一个给定的条件不会或不应该发生，那么它将必定发生。

一个简单的例子是一个程序需要用户输入一个必须找到其平方根的数值。当然，用户不应该输入负数，因为负数的平方根是虚数。然而，如果发生这种情况程序将如何处理？另一个永远应该考虑的可能情况是输入 0，特别是当作为算术操作的一部分使用时，尤其是除法操作。

人类的天性似乎是应该强迫程序员在解决一个问题时到键盘前面，诱使程序员跳过桌面检查这一阶段。有些时候这种自然的行为可减轻人脑对正在进行的实际过程的考虑。但是，当程序员具有了一些经验后，可很快认识到桌面检查节省时间的好处。

3.5 异常处理设计

错误处理是程序设计的基本组成部分，所以语言的结构反映出错误处理是完全正确的。ANSI C++ Committee 认识到，从语义上讲异常与循环退出和其他控制结构是不同的，他们推荐了一种特殊机制，处理语言中的这些情况。C++提供了如下的异常处理结构：

1. 定义异常。
2. 标记异常的发生。



3. 定义每一个异常分类的处理程序。

在实际使用中，一个子程序或代码块中可以指定一个异常处理程序，在其调用或间接调用的任何一个子程序中出现某一特殊的错误条件时，该异常处理程序被激活。对于不同类型的异常也可以指定不同的异常处理程序。C++翻译器必须实现将控制从信号发送者到一个处理程序的转换机制。

与错误返回码不同，程序员没有必要为每一个操作或每一个调用序列的子程序均指定一个单独的错误处理程序。另外，如果在调用序列中没有一个子程序为所发生的某一个异常定义处理程序，则程序中断。

异常处理程序的目的是在处理或结束该程序之前，清除该程序所需的状态。这种特性允许程序员将应用程序逻辑从错误处理代码中分离出来，使得每一个逻辑代码段的设计和实现非常清晰。

3.6 “请多多支持”

每次我们接近于一个道路建筑工地时都会看到这样的告示“Give'em a Hand”(请多多支持)。这一消息背后的含义到底是什么呢？请减慢速度，当心养路队员，他们在此是为了使您的生活更加幸福。文档对于不是这一代码编写者的一个程序员来说类似于这一简单消息。

本章试图分类错误，并推荐了检测错误的方法，也给出了完全避免错误的方法。然而，现实中程序员还是经常维护已有的算法。这就是说程序员的首要任务是掌握代码背后的逻辑：更新、移植和/或修复。通常情况下，一个程序员是否能够快速理解一个应用程序，取决于是否有好的文档。

这一最后的步骤在程序员的算法中常常被忽略，但这一步可能是比较重要的一步，特别是在商业编程领域。如果在定义问题、规划解决方案、编码、测试以及调试程序的各个方面已经做了很好的工作，那么建立文档应该是很容易的。最后的程序文档只是以上各步结果的记录。从最小程度来说，好的文档应该包括如下一些方面：

1. 问题定义的叙述性描述，包括输入类型、输出类型以及程序所使用的处理类型。
 2. 所有的设计限制和关于目标结构、内存限制、数据源、精度以及输出格式和设备的设想。
 3. 算法。
- 包含清晰注释摘要的源代码清单(见第 1 章)，程序内部的注释是整个文档的重要组成部分。每一个程序都应该在开始处包括一个注释，解释所做的工作、所使用的特殊算法以及对问题所做定义的概述。另外，还应该包括程序员的名字、程序编写的日期及最后的修改日期等内容。
4. 所希望的输入和所产生的输出实例。
 5. 关于测试和调试结果的信息。

6. 用户手册。

另外，正是对问题的描述(参见图 3-1)决定了一个算法的设计目标、定义程序的数据结构、效率、模型、代码风格、以及对文档的需求：无、少量或详细文档。

3.7 Microsoft Visual C++的帮助

快速浏览一下本地的 Barnes and Nobel 书店中，针对 Microsoft Visual C++的编程语言参考区，将很快证实如下的观点：当今的编译器文档可以堆满整个书架。从重量、复制开销及轻便性方面来看，Microsoft 已经将所有这些信息捆绑到了 MSDN CD Library 中(Microsoft Developer Network)，这真是一条好消息。

图 3-5 给出了 MSDN Library 的 Contents 标签集，用于跟踪 Visual C++对 C++关键字、操作符及标准数据类型的定义。这可能是读者了解一个编译器语法警告或错误消息方案的第一站。

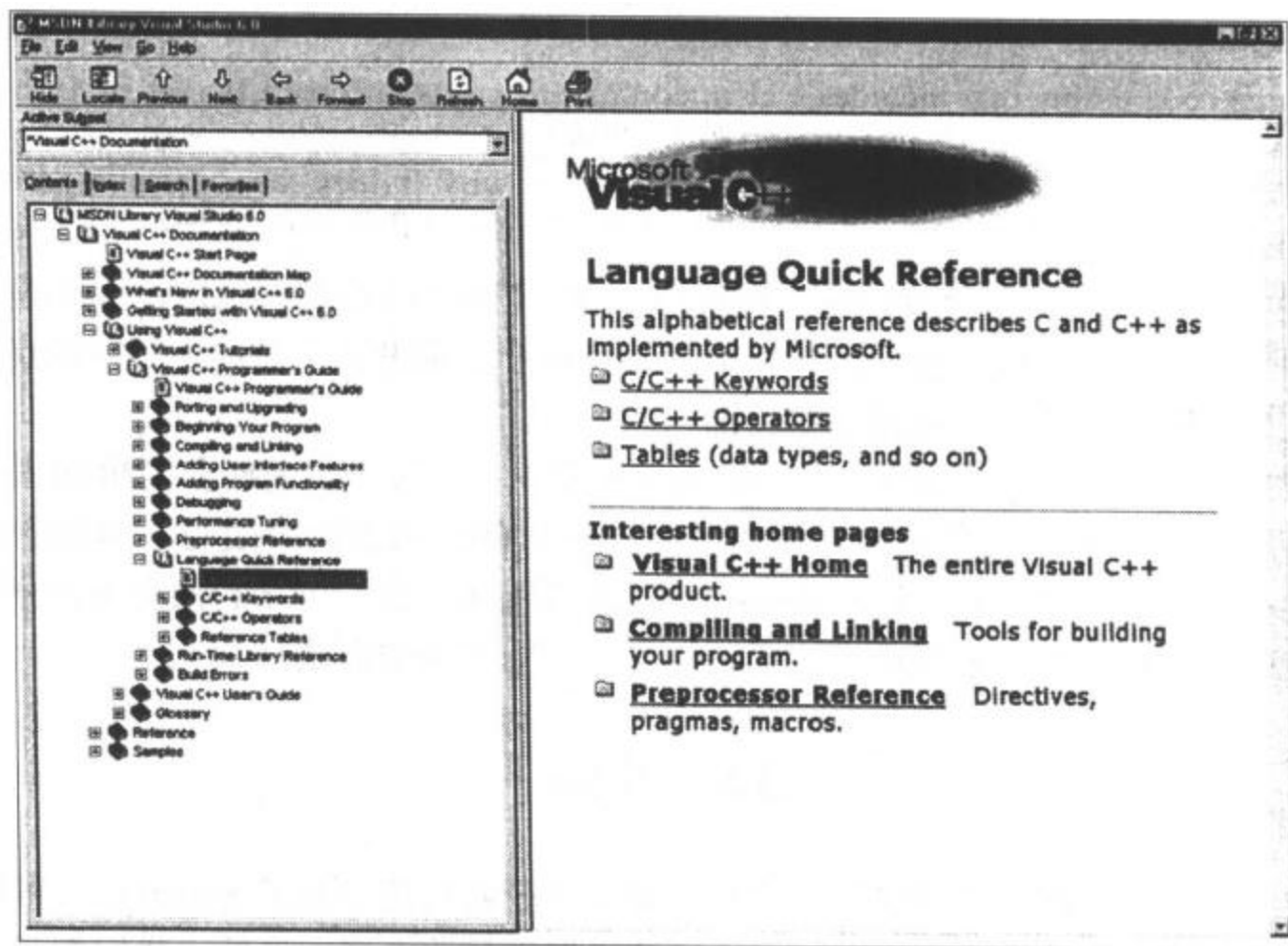


图 3-5 使用 Contents 标签定位 Language Quick Reference

图 3-6 给出了 MSDN Library 的 Index 标签集，其中给出的是若干个嵌套主题的详细 C++语言语法的开始处。与图 3-5 不同，图 3-5 给出的是如何学习 C++关键字和操作符的基础，而图 3-6 给出的是如何获得关于表达式、语句、类及 Microsoft 特有的 C++语法扩展的语法帮助。

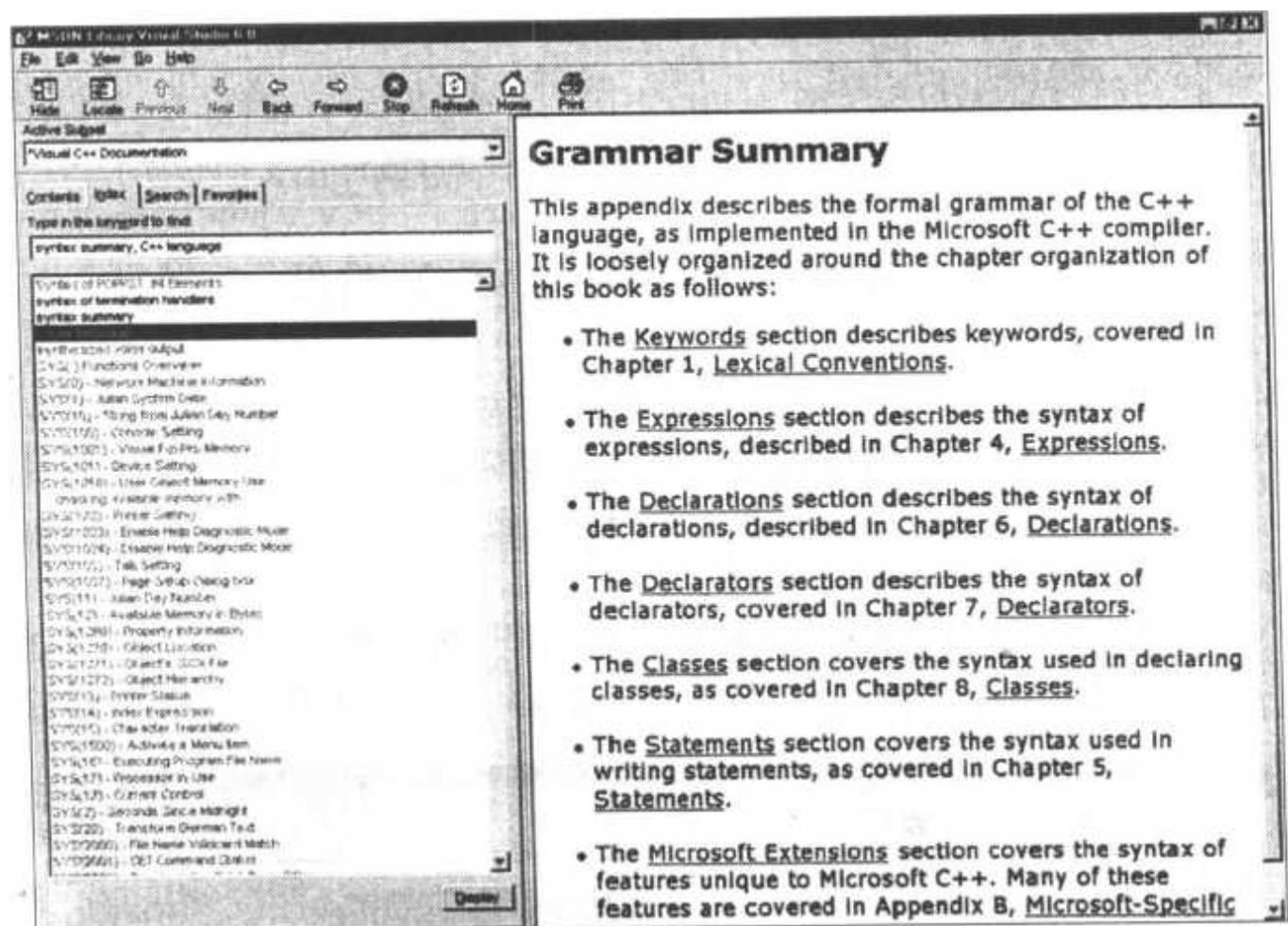


图 3-6 使用 MSDN Index 标签定位 C++语言的 Grammar Summary

人们说“智慧来自于在学校时的严厉敲打”。有经验的程序员都知道，编码是一个问题解决方案中最不重要的阶段。正确的问题定义、分析及逻辑设计，再加上严密的桌面检查和预演，才是产生即时可靠的应用程序的关键。

如果设想仅仅编译了三次就建立并运行应用程序——第一次是捕获简单的语法错误，第二次是测试逻辑，第三次是运行完整的产品——如何达到设计阶段呢？幸运的是，没有一个人将面对这样的情景，然而这却将焦点从快速编码转移到了可靠的预先编码分析和设计上来了。完成这一任务可以采用艰难的方法也可以采用容易的方法。

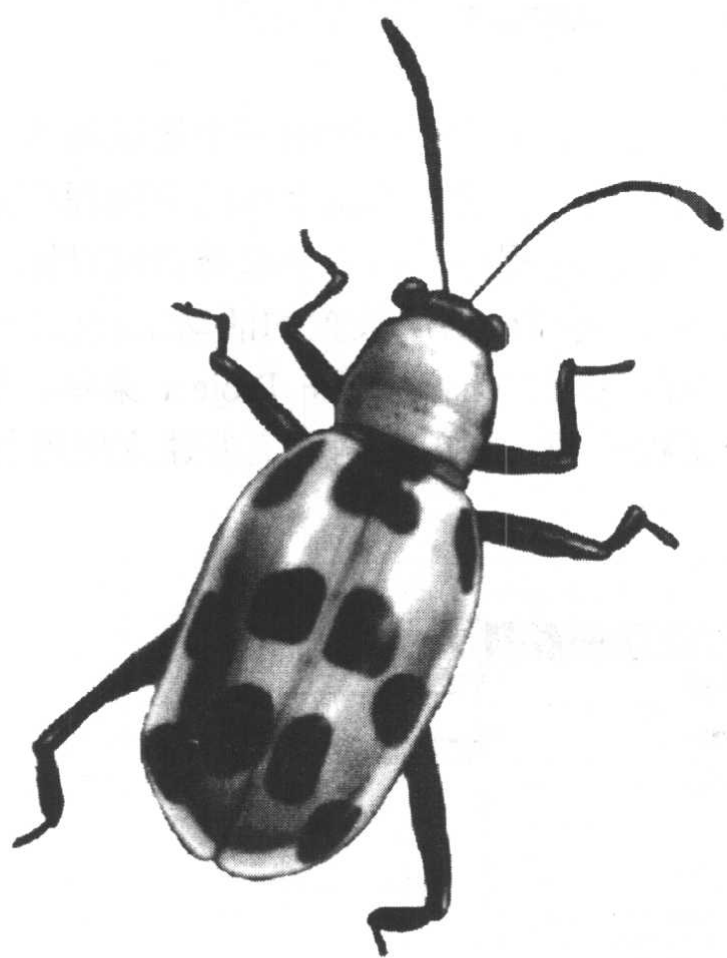
3.8 小结

本章中我们学习了产生程序错误的四种主要分类。我们还学习了 Visual C++ 可以通过跟踪错误码进入帮助文件的方法，详细描述特定的错误消息的内容。另外，我们学习或复习了这样的思想，即好的设计并无捷径，最后是“桌面检查！”

下一章中我们将介绍最基本的 Debugger 特性，这些特性在以后的日常工作中经常要用到。单就第 3 章中所提供的信息而言，可以很容易地节省我们 25% 到 50% 的应用程序调试时间。 ■

第 4 章

Debugger





Microsoft Visual Studio C++开发环境提供了多种工具，允许我们快速有效地跟踪源代码和程序组件中的错误。Debugger 的可视化界面包含有独特的菜单、窗口、对话框及电子表格字段。甚至可以访问 Windows 非常有用的拖放功能，与其他的 Debugger 组件转换信息。

本章将说明 Visual C++ Debugger 中所提供的最经常使用的菜单、窗口、对话框及电子表格字段，还将介绍程序员调整 Debugger 工具可以使用的定制选项。

错误监视

一定要确保在开始任何调试之前配置 Debugger 的选项。在许多情况下，忽略这一最关键的设置阶段将导致不必要的调试障碍，因为还没有定制 Debugger，提供应用程序专门的调试输出。

虽然大多数程序员最初都喜欢通过单击与应用程序交互，但在许多情况下，只要知道一些“热键”组合，可以节省大量时间，包括活动窗口与非活动窗口之间的切换，特定的 Debugger 选项可用之前窗口内部光标的重定位。

本章中我们将首先介绍如何使用下拉菜单或鼠标交互选择 Debugger 选项，然后将介绍可以使用的组合热键。为了便于参考，在本章结束时给出了一个热键简明列表，供读者快速查看有关信息。本章中所讨论的大多数技术及其他一些内容均将在以后各章的例子中使用。

4.1 确认 Debugger 可以使用

在运行 Visual C++ Debugger 之前，需要建立应用程序的一个调试版本。这种版本将导致编译器在目标文件中插入额外的符号调试信息。建立调试版本的应用程序是 Visual C++ 的缺省模式，但如果上一次建立过发行版本的应用程序，则需要选择调试目标。

在选择建立应用程序的目标类型之前，需要确认所设置的活动工程(只有在有多个工程打开的情况下才需要)。为确认活动的工程，单击 Visual C++ Project 菜单，选择 Set Active Project 选项(参见图 4-1)。如果有多个工程打开，则单击确认需要建立应用程序的工程。此处的例子中仅有一个“sample”。

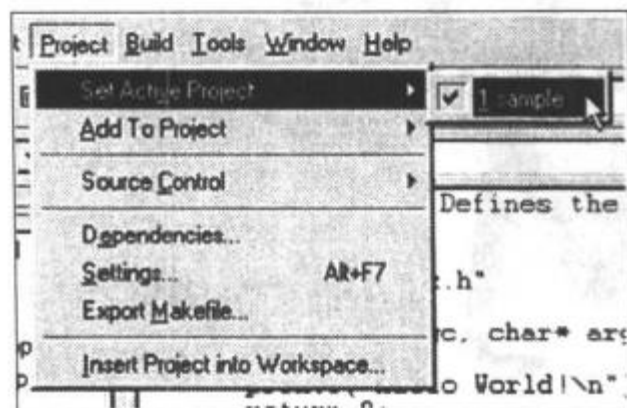


图 4-1 设置活动工程

图 4-2 给出了 Build 工具栏，其中在下拉列表中有两个目标类型可以选择。单击 Win32 Debug 选项。Build 工具栏上从左至右的选项为 Active Project 下拉列表(图 4-2 中的选择为 sample)，建立版本下拉列表(图 4-2 中的选择为 Win32 Debug)，之后是 Compile(编译)，Build(建立)，Stop Build(停止建立)，Execute(执行)，Go 以及 Insert/Remove Breakpoint(插入/删除断点)图标。Build 工具栏按钮在下一节中讨论。

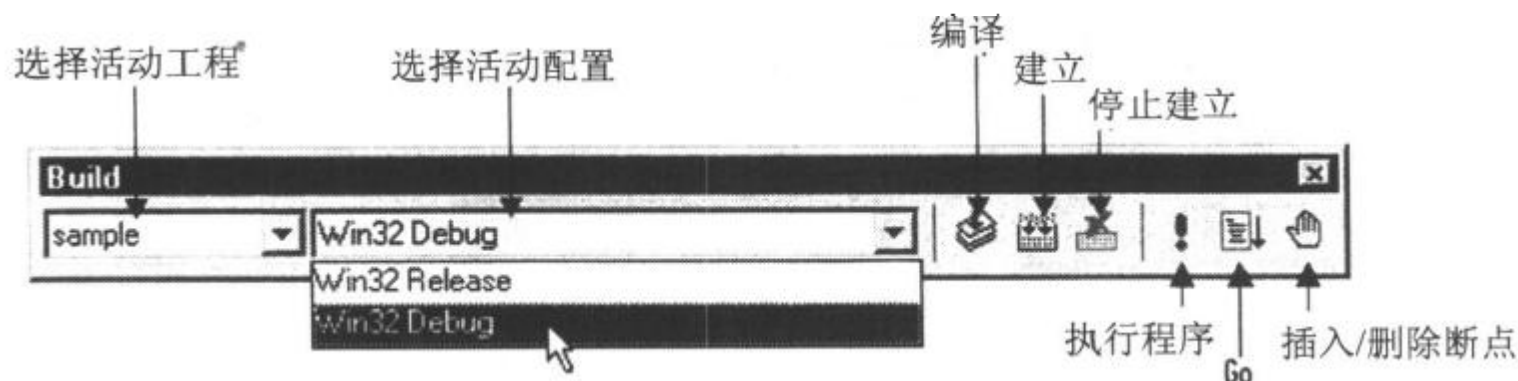


图 4-2 选择 Win32 Debug 建立版本

说明

如果看不见 Build 工具栏，则将鼠标置于 Visual C++ 菜单的任何位置(屏幕的顶部)，右击，选择 Build 复选框即可。

设计提示

Visual C++ 允许一个工作空间包含多个工程。确认选择了适当的活动工程，以产生所有必要的调试信息。

4.2 启动 Debugger

仅当有一个 Visual C++ 工作空间且工程或应用程序处于打开状态时，Debugger 选项才可以使用。

24x7

不能为 C/C++ 所支持的文件单独使用 Debugger——例如，在一个工程中仅包含初始化头文件。必须有一个可执行的源文件(例如，somesource.c 或 somesource.cpp)，并且该文件通过了编译和建立(build)，即有 0 个错误。

假设对一个 C/C++ 程序执行了一次有 0 个错误的编译，则从 Visual C++ 主 Build 菜单中选择一个选项，启动 Debugger，如图 4-3 所示(注意，可以在带有警告消息而不是错误消息的情况下运行 Debugger)。

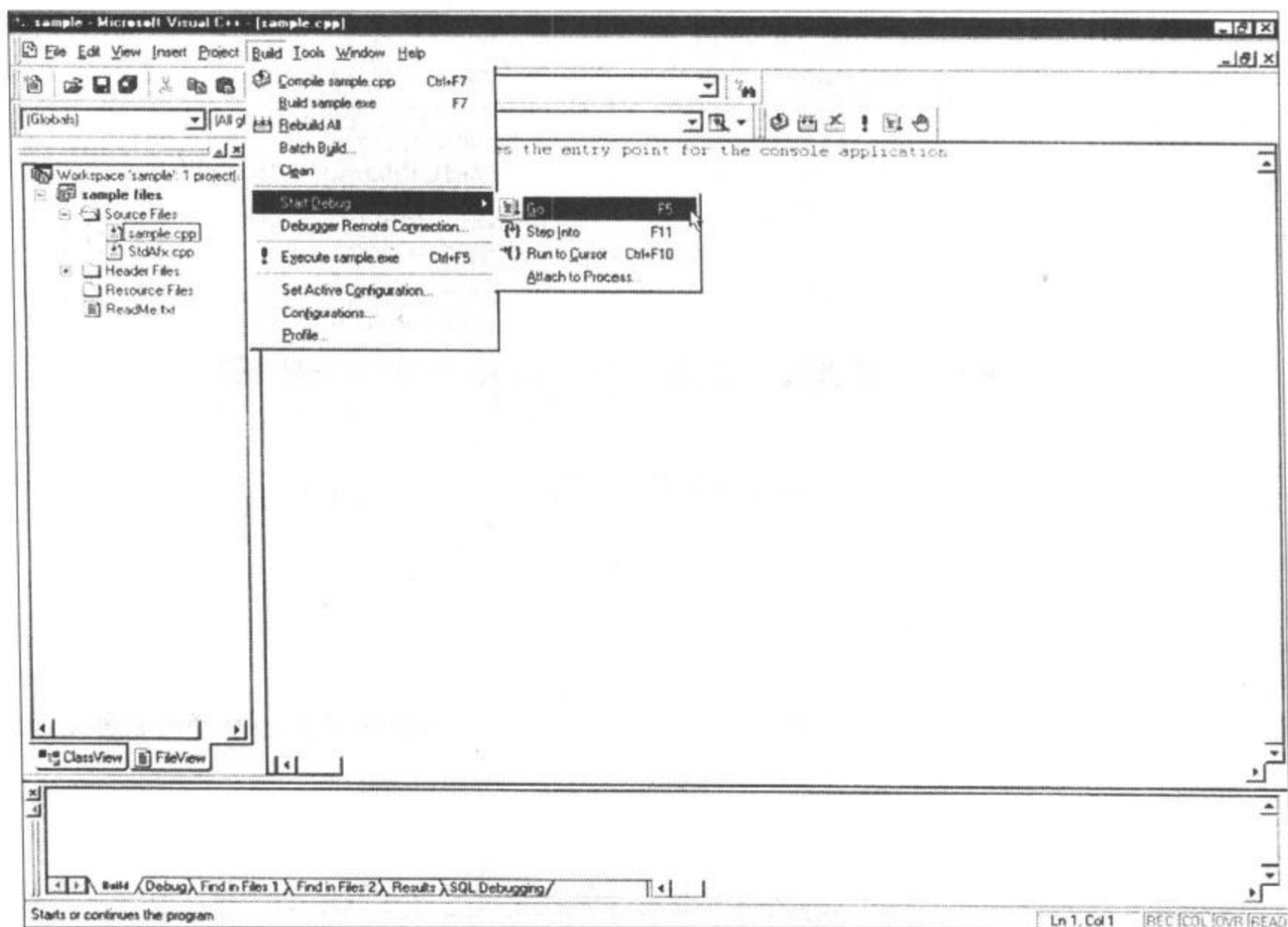


图 4-3 从 Build 菜单选项中启动 Debugger

通过单击 Build|Start Debug 选项，打开一个含有四个选项的列表，如图 4-4 所示。

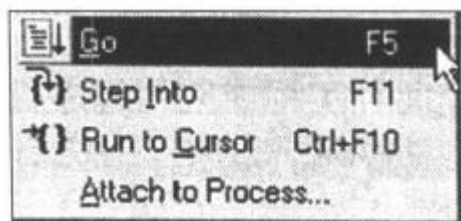


图 4-4 Build|Start Debug 选项菜单

选择这些选项中的任何一个均将使 Visual C++ 改变主 Build 菜单选项为 Debug 菜单，在 Debugger 运行期间(即使调试暂停在断点处)，该菜单出现在菜单栏中。

从主 Debug 菜单中我们可以控制程序的执行，并访问 QuickWatch 窗口。当 Debugger 处于非运行状态时，Debug 菜单替换为 Build 菜单。表 4-1 描述了所有的 Start Debug 选项及其相关的组合热键。

表 4-1 Start Debug 选项

Start Debug 选项	动作	热键
Go	启动 Debugger 并/或全速执行应用程序, 直到遇到一个断点或程序结束, 或直到应用程序暂停等待用户输入。与工具栏上的 Go 按钮类似	F5
Step Into	启动 Debugger 并/或逐行单步执行源文件。当所跟踪的语句包含一个函数或一个方法调用时, Step Into 进入所调用的子程序中	F11
Run to Cursor	启动 Debugger 并/或执行直到包含插入点光标的行。这一选项可以作为在插入点光标处设置常规断点一种选择	CTRL+F10
Attach to Process	附加 Debugger 到一个正在执行的程序, 然后调试者可以进入该程序, 如平常一样执行调试操作(这一选项是为高级用户准备的, 将在本章稍后讨论)	

还有一个组合热键 F10 没有在该初始菜单中列出, F10 表示的是 Step Over。通过按 F10, 可以启动 Debugger 并/或逐行执行程序。

4.2.1 Step Into 和 Step Over 的区别

F10 和 F11 之间的区别只有在将要被执行的语句是一个子程序(函数或方法)调用并且按了下一个 F10 或 F11 时才能显示。正如 Step Over 所暗示的那样, 在一个调用语句上按 F10 将指示 Debugger 全速执行所调用的子程序, 并暂停在调用过程中该调用下面的代码语句处。

另一方面，当下一个 F10 或 F11 将调用一个子程序时，F11 或 Step Into 将跟踪到被调用子程序(函数或方法)内部，并使 Debugger 暂停在被调用子程序内部。

24x7

对于标准 C/C++ 子程序应该按 F10 而不是 F11(Step Into)，以避免 Debugger 调试编译器所提供的代码。

4.2.2 Go

有时选择 F10(Step Over)或 F11(Step Into)均浪费时间,例如,对于许多重用了以前编写和调试好的算法的程序,单步调试老的算法将是浪费时间。在这些情况下,可选择 Go(F5)选项。Go 全速执行程序直到遇到第一个断点(本章稍后讨论),或重复按 F5 所遇到的下一个断点,或到程序结束。

设计提示

Go 选项最好的用途之一是有用地调试循环。将断点设置在循环体内，重复按 F5 将指示 Debugger 全速执行循环体。当测试正确的循环迭代时可使用这种方法。

4.2.3 Run to Cursor

Run to Cursor 选项(CTRL+F10)类似于 Go 命令, 只是 Run to Cursor 不需要事先定义断



点(本章稍后讨论)。为使用 Run to Cursor, 只要将插入光标符 I 移动到算法中需要开始调试的语句, 然后按 CTRL+F10 即可。

4.3 理解 Debugger 工具栏图标

当开始调试程序后, 使用前一节中所描述的任何一种方法, 均将看到缺省的 Debugger 工具栏。Debugger 工具栏如图 4-5 所示。

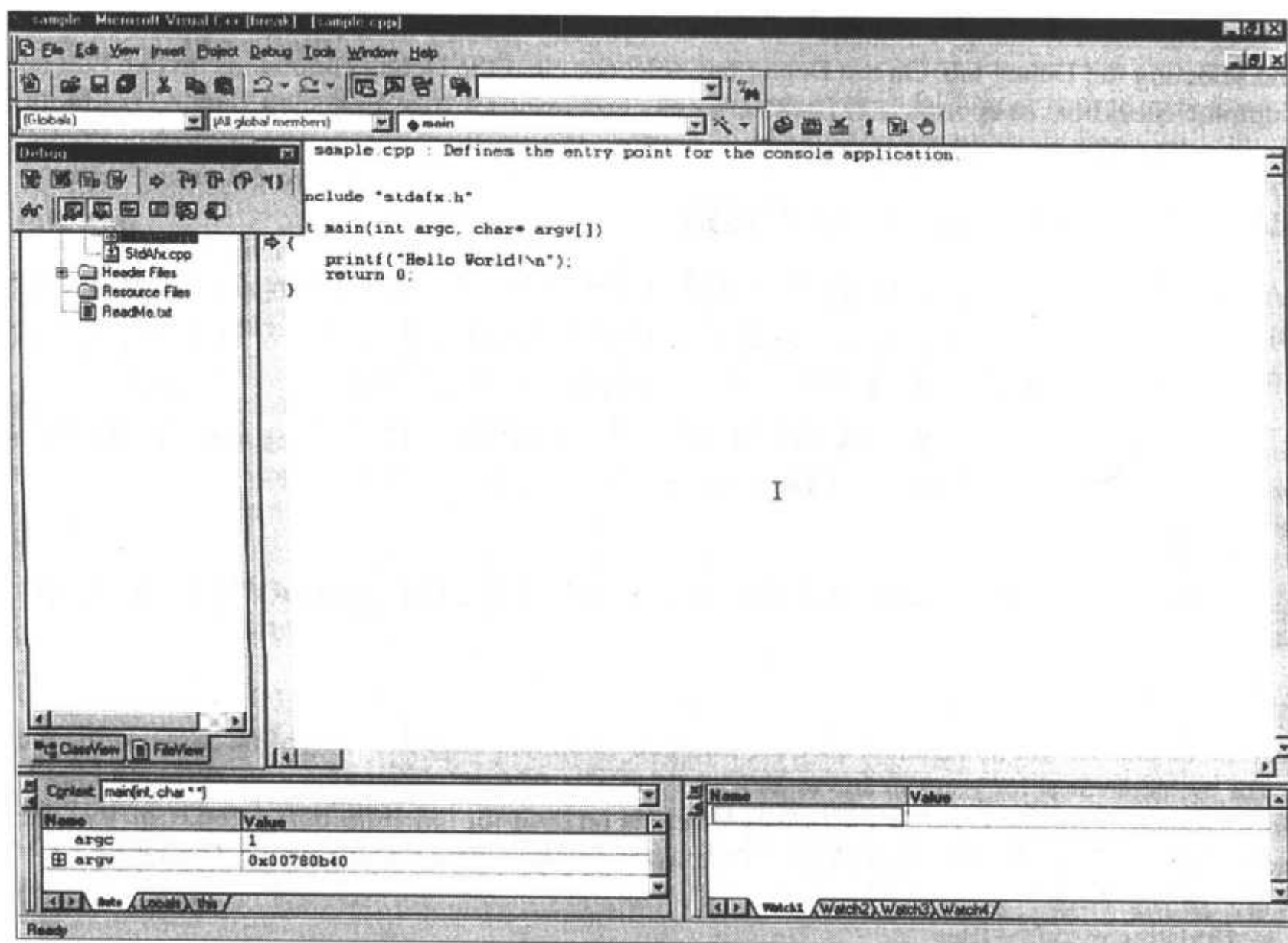


图 4-5 初始 Debugger 工具栏

图 4-5 中给出的 Debugger 工具栏显示在 Workspace 窗格之上。如果用户不喜欢其缺省位置, 只要单击该工具栏后将其拖动到所需位置即可。当然, 也可以将该工具栏拖动到主菜单位置并将图标停放在其中。

图 4-6 给出了该工具栏中图标的解析视图。如下的讨论将说明每一个图标, 从第一行的

左上角的图标开始，向右至最右边的图标，然后再从第二行继续。

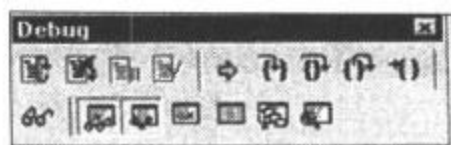


图 4-6 Debugger 工具栏

4.3.1 Restart

Restart 图标(CTRL+SHIFT+F5)指示 Visual C++，用户需要从开始处调试程序，而不是从当前所跟踪的位置开始调试。

4.3.2 Stop Debugging

如果在调试阶段的任何点上认识到需要更新工程或工作空间，则只有当首先使用 Stop Debugging(SHIFT+F5)结束 Debugger 后，才能返回到 Build 菜单选项。

4.3.3 Break Execution

使用 Break Execution 按钮可以在当前点上挂起程序的执行。

4.3.4 Apply Code Changes、Edit and Continue

使用 Apply Code Changes(ALT+F10)，可以在程序正在调试时修改源代码，这是 Visual C++ 6.0 的新特性。用户可以在程序正在 Debugger 下运行或挂起时应用修改的代码。为应用修改的代码到正在调试的程序，单击 Debug 菜单的 Apply Code Changes 即可。

使用 Edit and Continue，在对一个挂起的程序选择 Go 或 Step 命令后，可以自动应用所做的修改。如果愿意，则可以关闭自动 Edit and Continue 特性(如果关闭该自动特性，仍可以手工使用 Apply Code Changes 应用修改的代码)。

通过单击 Tools|Options 菜单，然后选择 Debug 标签，打开或关闭 Automatic Edit and Continue。在 Debug 标签中，选择和清除 Debug 命令将正确激活 Edit and Continue 复选框。由于 Debug 命令激活的 Edit and Continue 是一个工具选项，而不是一个工程选项，所以改变该设置将影响到所有工程。在修改这一设置后无须重新建立应用程序，甚至在调试时也可以修改该设置。

Program Database 保存 Edit and Continue 代码修改所需的信息，Visual C++ 选择 Project Settings 对话框中的适当选项(Program Database for Edit and Continue)。如果修改该选项(例如，为了使用 C7 兼容的调试信息)，Edit and Continue 将关闭。



如果需要为特定工程关闭 Edit and Continue, 则完全可以。选择 Project|Settings 菜单选项, 然后选择 C/C++ 标签, 选择 General 分类。下一步, 选择 Debug Info 下拉列表。然后选择 Program Database for Edit and Continue, 以打开或关闭该选项, 单击 OK 后重新建立该应用程序。

设计提示

如果正在调试优化的代码, 则不要打开 Edit and Continue。Edit and Continue 与优化选项是不兼容的, 如果打开该选项, 则将产生编译错误。

4.3.4.1 不能编辑并继续的语句类型

由于 Edit and Continue 所使用的内部机制原因, 对于可以在一个活动函数添加的新变量的总长度有一个 64 字节的限制, 一个活动函数是指当前处于调用堆栈中的函数。另一方面, 对于当前未处于调用堆栈中的函数则不存在任何限制。以下列表类举了 Edit and Continue 不能处理的代码修改类型:

- 更新资源文件。
- 修改只读文件中的代码。
- 修改使用 /O1、/O2、/Og、/Ox、/Ob1 或 /Ob2 优化选项优化过的代码。
- 修改异常处理块。
- 修改数据类型, 包括类、结构、联合或枚举的定义。
- 添加新数据类型。
- 删除函数或修改函数原型。
- 修改全局变量或静态代码。
- 更新从另一台机器复制而来并且未在本本地建立的可执行程序。

如果执行了这些修改之一, 然后试图应用这种代码修改, 则将在 Output 窗口中出现一条错误消息。

4.3.5 Show Next Statement

该选项(ALT+NUM*)显示程序代码中的下一条语句。如果源代码找不到, 则 Show Next Statement 在 Disassembly 窗口内显示语句。

4.3.6 Step Into

当选择 Step Into(F11)并且正在跟踪的语句是一个子程序调用(函数或方法)时, 该选项单步进入所调用的子程序。



4.3.7 Step Over

当选择 Step Over(F10)并且正在跟踪的语句是一个子程序调用(函数或方法)时, 则该选项通过全速执行所调用的子程序, 单步通过所调用子程序。Debugger 停留在子程序调用下面的语句上。

4.3.8 Step Out

Step Out(SHIFT+F11)使得 Debugger 切换回全速执行到被调用函数结束并停留在调用该子程序紧后面的一条指令上。当确认当前子程序中没有程序错误时, 可使用该命令快速执行该子程序。

4.3.9 Run to Cursor

Run to Cursor 选项(CTRL+F10)与 Go 命令类似, 只是 Run to Cursor 不需要事先定义断点(本章稍后讨论)。为了使用 Run to Cursor, 只要将 I 光标移动到源文件中需要开始调试该算法的语句处, 然后按 CTRL+F10 即可。

4.3.10 QuickWatch

QuickWatch(SHIFT+F9)显示 QuickWatch 窗口, 在该窗口中可以计算表达式的值。

设计提示

现在, 可以使用符号 MM0-MM7 在 Watch 和 QuickWatch 窗口内显示 MMX 寄存器的内容。MMX 寄存器是 64 位整数寄存器, 并且可以在所有 x86 机器上显示, 无论这些机器是否支持 MMX 指令。

4.3.11 Watch

Watch 图标打开 Watch 窗口, 该窗口包含该应用程序的变量名及其当前值, 以及所有选择的表达式。

4.3.12 Variables

Variables 按钮打开 Variables 窗口, 该窗口包含关于当前和前面的语句中所使用的变量和返回值(在 Auto 标签中)。当前函数中的局部变量(在 Local 标签中), 以及由 this 所打印的对象(在 This 标签中)。



4.3.13 Registers

Registers 按钮打开 Debugger 的 Registers 窗口，显示微处理器的一般用途寄存器和 CPU 状态寄存器。

4.3.14 Memory

使用 Memory 按钮打开 Memory 窗口，显示该应用程序的当前内存内容。

4.3.15 Call Stack

当单击 Call Stack 按钮后，Debugger 打开 Call Stack，列表显示所有未返回的被调用的子程序名。

4.3.16 Disassembly

Disassembly 图标打开一个包含汇编语言代码的窗口，其中的汇编语言代码来自编译后程序的反汇编。

4.3.17 Debugger Toolbar Menu Equivalents

当然，如果选择关闭 Debugger 工具栏，则所有相同的 Debugger 选项将可以通过主 Debug 菜单(参见图 4-7)获得。切记，当 Debugger 处于活动状态时，Visual C++ 主 Build 菜单选项将其标题改变为 Debug。

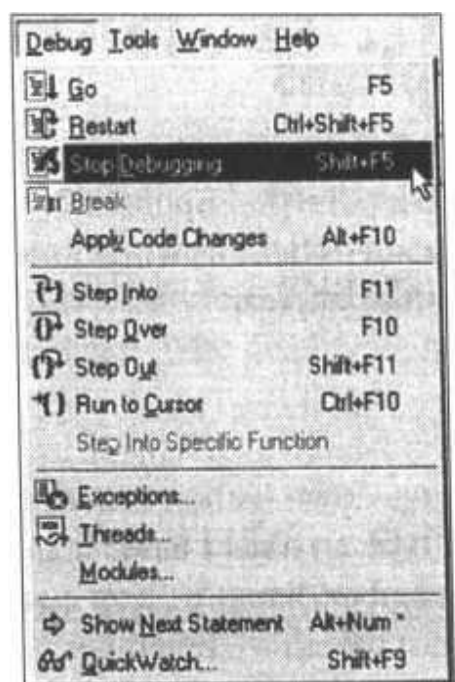


图 4-7 Debug 菜单等价于 Debug 工具栏图标



4.4 其他 Debug 菜单选项

实际上，如果仔细检查图 4-7，即可发现 Debug 工具栏上有一些其他调试选项并不是立即可以使用，包括 Step Into Specific Function(在图 4-7 中为暗淡显示)、Exceptions、Treads 及 Modules。

4.4.1 Step Into Specific Function

Step Into Specific Function 选项单步通过程序中的指令，并进入指定的函数调用。该功能对于函数的嵌套层不限。

4.4.2 Exceptions

Exceptions 选项打开 Exceptions 对话框，显示需要处理的异常列表，用户可以修改、删除或添加所需的异常。该列表保存在以.dsw 为扩展名的一个文件中，与工程同时存在。

4.4.3 Threads

Threads 菜单选项显示 Threads 对话框，该对话框允许挂起程序线程、恢复或设置焦点。

4.4.4 Modules

Modules 窗口列表显示所有由应用程序所加载的 Dynamic Link Libraries(DLLs)。列表的顺序为其加载的顺序。通过单击列表上面的按钮，可以通过名字、内存地址、路径或加载顺序排序该列表。

4.5 本地菜单 Debugger 选项

图 4-8 显示了可以使用的 Debugger 选项列表，当 Debugger 处于活动状态时从 Edit 窗口中可以得到这些选项。这是一个弹出菜单，通过在 Visual Studio C++ Edit 窗口中右击可弹出该菜单。如下的讨论只包括那些本章的前面未讨论过的菜单选项。

4.5.1 List Members

List Members 选项显示一个下拉列表框，其中包含了所选择对象的属性和方法。在使用 object.member 语法输入一个对象的名字时，当输入句点成员操作符后列表即可出现。可以通过输入成员名字的前面一个或多个字母，直到所需要的成员高亮显示为止。也可以使用 UP-ARROW、DOWN-ARROW、PAGEUP、PAGEDOWN、CTRL+PAGEUP 或 CTRL+PAGEDOWN 键，导航遍历该列表。

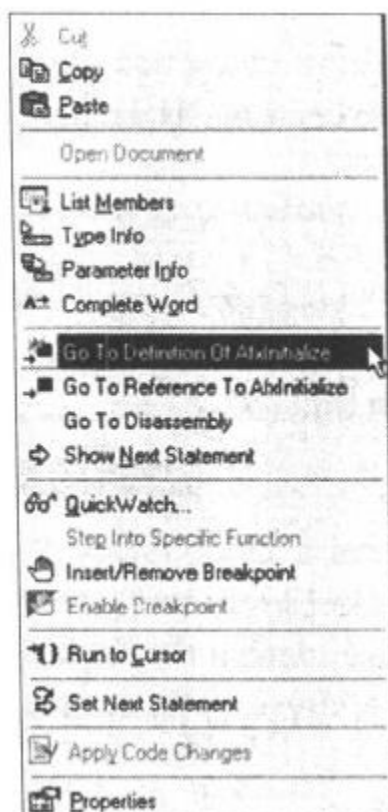


图 4-8 在 Edit 窗口中可以使用的 Debugger 选项

4.5.2 Type Info

当选择 Type Info 选项时，Visual Studio 将显示在 Edit 窗口中所选择的一个变量、函数或方法的语法。

4.5.3 Parameter Information

通过单击 Parameter Information 选项，可以快速访问关于一个函数的参数或语句的信息。如果有一个包含函数调用作为其参数的函数或语句，则选择 Parameter Info 可提供关于第一个函数的信息。Quick Info 提供关于每一个内嵌函数的信息。当 Parameter Info 窗口可见时，输入一个逗号(,)将使下一个参数黑体显示。要撤消 Parameter Info，可按如下方法执行：

- 单击 Edit 窗口中的任何位置。
- 输入所需的全部参数。
- 在没有使用所有的可选参数时结束该函数。
- 按 ESC。

4.5.4 Complete Word

Complete Word 选项填充正在输入的单词的其余部分，只要所输入的字符足以使编辑器识别所需要的单词即可。

4.5.5 Go To Definition/Reference

Go To Definition 或 Go To Reference 快速显示一个符号的定义或引用。开发环境将打开包含首次定义或引用的源文件，并高亮显示该符号。为查看下一个定义或引用，单击工具栏上的 Next Definition/Reference 按钮。

4.5.6 Go To Disassembly

参见本章前面的“Disassembly”一节。

4.5.7 Insert/Remove Breakpoint

在本章前面讨论 Go(F5) Debugger 命令时，我们学习了 Go 命令将全速执行程序到一个断点，或者如果没有断点存在时执行到程序结束。Insert/Remove Breakpoint 选项是一个开关菜单项，它可以打开或关闭一个断点。为设置一个断点，只要将光标放置于 Edit 窗口中需要 Debugger 暂停的代码行，然后选择该选项即可。

4.6 Debugger 窗口

根据正在调试的 Windows 应用程序的类型和个人的喜好，Visual C++ 提供了一个针对于当前任务的 Debugger 窗口。用户在调试时可以使用 View 菜单访问这些窗口。表 4-2 列出了 Debugger 窗口，并描述了这些窗口所显示的信息。

表 4-2 标准 Visual C++ Debugger 窗口

Debugger 窗口	显示
Output	Output 窗口显示关于建立程序过程的信息，包括编译器、连接器或建立工具的错误，其中还显示来自 OutputDebugString 函数或 afxDump 类库的输出信息，线程终止代码，加载符号标志信息，以及首次异常标志信息
Watch	Watch 窗口输出变量和表达式的名字和值
Variables	Variables 窗口的输出信息包括：当前和前面的语句中所使用的变量信息和函数的返回值信息(在 Auto 标签中)，当前函数的局部变量信息(在 Locals 标签中)，以及由 this 所指的对象信息(在 This 标签中)
Registers	Registers 窗口显示一般用途的寄存器和 CPU 状态寄存器的当前内容
Memory	Memory 窗口显示当前内存的内容
Call Stack	Call Stack 窗口显示所有尚未返回的函数调用的堆栈
Disassembly	Disassembly 窗口显示来自编译后程序的反汇编的汇编语言代码

当一个窗口处于悬浮模式时，用于可以调整该窗口的尺寸或最小化该窗口，增加其他窗口的可视区域。用户可以从任何一个 Debugger 窗口复制信息。



设计提示

用户只能在 Output 窗口中打印信息。

通过使用 Options 对话框中的 Debug 标签,可以定制这些窗口的格式和其他选项(从 Tools 菜单中进入)。

4.6.1 Trace 窗口

当激活 Debugger 时, Edit 窗口改变为 Trace 窗口, Trace 窗口允许用户在代码执行时查看代码行。

4.6.1.1 跟踪对象代码

当调试面向对象的源代码时, Trace 窗口与 Object 列表中所显示的对象相关联。Object 列表中包含了最高层父容器中其代码当前正在运行的所有对象。

4.6.1.2 跟踪面向过程的代码

当跟踪函数或方法时, Trace 窗口中的代码与 Procedure 列表中所显示的方法或事件相关联。Procedure 列表中包含了 Object 列表中、含有与之相关代码的对象的所有方法。

4.6.2 Watch 窗口

Watch 窗口显示表达式及其当前值,并在表达式处设置断点。在 Watch 中,用户可以输入表达式,将其添加到活动的观察表达式网格,该表达式位于 Watch 窗口下方。

Name 分类显示了当前观察表达式的名字,之后是 Value 列,其中显示当前观察表达式的值。可以使用 Type 分类观察表示当前观察表达式数据类型的字符。另外, Watch 窗口允许选择表达式、删除表达式或在表达式中添加断点。

4.7 View 菜单和 Debugger 窗口

通过首先单击主 View 或 View/Debug Window 菜单选项,然后单击窗口类型之一,可以激活一个隐藏或压缩了的 Visual C++ Debugger 窗口。下面分别介绍这些窗口。

4.7.1 Workspace

Workspace 窗格或窗口显示当前工程的源文件、类对象及资源文件。

错误监视

如果通过将鼠标放置在窗格的右边界上,并将其向最左边移动,压缩了这一窗格,那

么这一菜单命令将不重新显示该窗口。必须手工定位压缩窗格分界线后重新扩展该窗口。

4.7.2 Output

这一选项激活 Debug Output 窗口。注意，如果该窗口停放在某处，则并没有可视信息表明已经激活，尽管相关的菜单项为打开或关闭。

4.7.2.1 清除 Output 窗口

该窗口删除所有显示在 Debug Output 窗口中的文本(Debug Output 窗口仅在 Output 窗口快捷菜单中活动——使用右击 Output 窗口激活)。

4.8 以不同的数据类型查看观察变量

在以后各章中，我们将开始使用前面几节中描述的 Debugger 选项。然而在考虑各种 Debugger 窗口的输出之前，应该熟悉 4-3 中列出的格式化符号。

表 4-3 Debugger 显示格式化符号

符号	格式	值	表示
d,l	有符号(signed)十进制整数	0xF000F061	-268373911
U	无符号(unsigned)十进制整数	0x006	102
O	无符号八进制整数	0xF064	0170144
x,X	十六进制整数	70148(十进制)	0x0000112.4
l,h	用于 d、i、u、o、x、X 的长型(long)或短型(short)前缀	00406040,hx	0x0c20
F	有符号浮点数	5./2.	2.500000
E	有符号科学表示法	5./2.	2.500000e+000
G	有符号浮点数或有符号科学标识法，取其较短者	5./2.	2.5
C	单字符	0x9966	'f'
S	字符串	0x0012fde8	"The String"
su	统一码字符串		"The String"
St	统一码字符串或 ANSI 字符串，取决于统一码字符串(Unicode Strings)的设置		
Hr	HRESULT 或 Win32 错误码	0x00000000L	S_OK
wc	窗口类标志	0x00000040	WC_DEFAULTCHAR
wm	Windows 消息码	0x0010	WM_CLOSE

这些符号也允许改变 QuickWatch 和 Watch 窗口中的变量显示格式。
Watch 窗口允许改变对变量数据类型的解释。要使用格式化符号，只要输入变量的名字，



再输入一个逗号和适当的格式符号即可。例如，如果 *hexNumber* 的值为 *0x0041*，需要以字符方式显示该值，则在 Watch 窗口的 Name 列中输入 *hexNumber, c* 即可。当输入 Enter 键后，则字符形式的值出现：

```
hexNumber, c = 'A'
```

表 4-4 中列出了可以在 Watch 窗口中使用，用来格式化内存单元中内容的格式化符号。

表 4-4 内存格式化符号

符号	格式
Ma	64 ASCII 字符
M	十六进制的 16 字节，后跟 16 ASCII 字符
Mb	十六进制的 16 字节，后跟 16 ASCII 字符
Mw	8 字
Md	4 双字
Mq	4 四字
Mu	2 字节字符(统一码)

也可以在所输入的语句中使用内存单元格式化符号，所输入的语句用求一个单元(内存地址)的值。为了将一个字符数组的值显示为一个字符串，在数组名的前面置一个 & 符号，即 &ArrayName 即可。也可以在表达式后跟一个格式化字符，如 &ArrayName, x。

最后，监视特定内存单元中的值或一个寄存器所指向的值时，使用如下所述的操作符 BY、WO 或 DW：

- **BY** 返回所指向字节的内容。
- **DW** 返回所指向双字的内容。
- **WO** 返回所指向字的内容。

要使用这些操作符，只要在操作符后指定变量、寄存器或常量的名字，如 BY *Achar* 即可。此时，Watch 窗口显示变量中所包含的地址处的字节、字或双字。

也可以使用上下文操作符 {}，显示任何单元中的值。要在 QuickWatch 或 Watch 窗口中显示一个 Unicode 字符串，可使用 su 格式指示符。要在 QuickWatch 或 Watch 窗口中以 Unicode 字符显示数据字节，使用 mu 格式指示符。

设计提示

格式化符号与数组、结构、指针及对象一起使用时，仅以非扩展变量形式使用。如果扩展了变量，则指定的格式将影响到所有成员。然而，不允许对单个成员使用格式化符号。

4.9 打开 Just-in-Time 调式

还有一个可以使用的选项是 Microsoft 的 Just-in-time 调试。使用 Just-in-time 调试，应用程序可以在 Visual C++ 开发环境之外运行，直到发生一个错误。当遇到一个程序错误时，Just-in-time 调试自动加载 Visual C++ Debugger。为打开 Just-in-time 调试，执行如下步骤：

1. 单击 Tools 菜单中的 Options。
2. 选择 Debug 标签。
3. 选择 Just-in-time debugging 复选框。
4. 单击 OK。
5. 在 Build 菜单中选择 Build myApp.exe。

说明

Windows NT 程序员必须具有管理权限，才能打开 Just-in-time 调试。

4.10 Options 窗口中的 Debug 标签

Debugger 的输出显示格式非常灵活。虽然可以在空闲时修改许多选项，但有时更喜欢使用一致的输出格式。图 4-9 显示了 Debug 标签中预定义这些格式可以使用的选项。通过单击 Tools 菜单后选择 Options，可以访问 Debug 标签。下面分别讨论其中的选项。

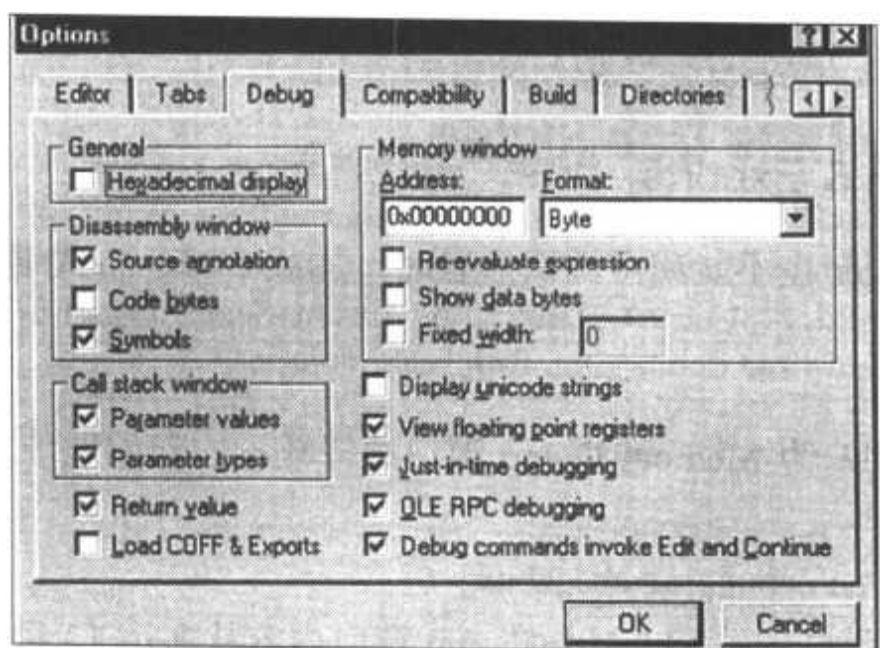


图 4-9 Debug 标签



4.10.1 Hexadecimal Display

这一选项以十六进制格式显示所有值，并以十六进制解析所有的用户和对话框输入。当选择了该选项时，可以使用 *On* 前缀输入十进制值，如 *On123456789*。

4.10.2 Source Annotation

这一选项同时显示源代码及其汇编语言翻译。

4.10.3 Code Bytes

这一选项显示与每一个汇编语言指令相关联的字节。

4.10.4 Symbols

Symbols 选项在 Disassembly 窗口中显示地址的符号名。

4.10.5 Parameter Values

Parameter values 选项显示调用堆栈中传递给一个子程序调用的参数的各参数值。

4.10.6 Parameter Types

该选项显示调用堆栈中传递给一个子程序调用的参数的类型信息。

4.10.7 Return Value

该选项在 Vairalbes 窗口中显示子程序的返回值。

4.10.8 Load COEF & Exports

该选项使得 Debugger 在不能获得调试信息时，加载 COEF 格式调试信息或 DLL 导出。

4.10.9 Address

Address 选项允许用户为 Memory Dump 窗口选择缺省的起始地址。

4.10.10 Format

该选项允许用户对所显示的 Memory Dump 窗口内容从多达 13 种数据类型映射中选择，如 ASCII、Byte、Long、Real 及 Int64。

4.10.11 Re-evaluate Expression

该选项动态重定位 Memory Dump 窗口的内容，当该选项处于活动状态，并且已经在 Memory Dump 窗口中输入了一个指针变量的名字时，则每次该指针变量的地址内容改变时，该窗口的内容将得到更新。当该选项处于关闭状态时，Memory Dump 窗口的内容将不改变，即使指针变量的地址内容确实已经改变。

4.10.12 Show Data Bytes

该选项以原始字节和所选择的格式显示数据。

4.10.13 Fixed Width

Fixed width 选项以固定宽度格式显示 Memory Dump 窗口的内容，此处所输入的值与所选择的格式类型一起发生作用。

4.10.14 Display Unicode Strings

该选项显示 Unicode 格式的字符串。当调试输入输出使用多种语言格式的文本数据的应用程序时，该选项非常有用。

4.10.15 View Floating Point Registers

该选项在 Registers 窗口中显示浮点寄存器。

4.10.16 Just-in-Time Debugging

当一个在 Visual Studio 开发环境之外运行的应用程序遇到运行错误时，该选项打开自动激活 Visual C++ Debugger 功能。

4.10.17 OLE RPC Debugging

OLE(Object Linking and Embedding) RPC(Remote Procedure Calls) debugging 选项允许调试远程过程调用。

4.10.18 Debug Commands Invoke Edit and Continue

使用该选项将允许许多但不是全部代码修改，在下一次使用 Debugger 命令如 Go、Step Into、Step Over 或 Run 时，自动应用于源代码。



4.11 键盘映射

通过选择 Visual C++ Help|Keyboard Map 选项，可以显示当前键盘的快捷键，包括定制键设置和编辑器模拟用键，如图 4-10 所示。



图 4-10 选择 Visual C++ Help|Keyboard Map 选项

关于 Debugger 选项的键盘映射在图 4-11 中给出。

The 'Help Keyboard' dialog box is shown with the 'Debug' category selected. It contains a table with the following data:

Categ	Command	Keys	Description
Debug	ApplyCodeChanges	Alt+F10	Applies code changes made to C/C++ source files while debugging
Debug	DebugBreak		Stops program execution; breaks into the debugger
Debug	DebugDisableAllBreakpoints		Disables all breakpoints
Debug	DebugEnableBreakpoint	Ctrl+F9	Enables or disables a breakpoint
Debug	DebugExceptions		Edits debug actions taken when an exception occurs
Debug	DebugGo	F5	Starts or continues the program
Debug	DebugHexadecimalDisplay		Toggles between decimal and hexadecimal format
Debug	DebugMemoryNextFormat	Alt+F11	Switches the memory window to the next display format
Debug	DebugMemoryPrevFormat	Alt+Shift+F11	Switches the memory window to the previous display format
Debug	DebugModules		Shows modules currently loaded
Debug	DebugQuickWatch	Shift+F9	Performs immediate evaluation of variables and expressions
Debug	DebugRemoveAllBreakpoints	Ctrl+Shift+F9	Removes all breakpoints
Debug	DebugRestart	Ctrl+Shift+F5	Restarts the program
Debug	DebugRunToCursor	Ctrl+F10	Runs the program to the line containing the cursor
Debug	DebugSetNextStatement	Ctrl+Shift+F10	Sets the instruction pointer to the line containing the cursor
Debug	DebugShowNextStatement	Alt+Num *	Displays the source line for the instruction pointer
Debug	DebugStepInto	F11	Steps into the next statement
Debug	DebugStepIntoSpecificFunction		Steps into the selected function
Debug	DebugStepOut	Shift+F11	Steps out of the current function
Debug	DebugStepOver	F10	Steps over the next statement
Debug	DebugStepOverSource		Steps over the next source level statement
Debug	DebugStopDebugging	Shift+F5	Stops debugging the program
Debug	DebugThreads		Sets the debuggee's thread attributes
Debug	DebugToggleBreakpoint	F9	Inserts or removes a breakpoint
Debug	DebugToggleMixedMode	Ctrl+F11	Switches between the source view and the disassembly view for this instruction
Debug	UpdateImageToggle		Applies code changes made to C/C++ source files while debugging

图 4-11 Debugger 选项的键盘映射



键盘映射选项可带来许多方便，特别是当需要使用 Debugger 的功能，而自己又想不起如何激活该功能时。对于当今具有许多选项和菜单的开发环境来说，很容易忘记在何菜单、按钮或对话框中有自己所需要的内容。通过使用 Help Keyboard 窗口，可以快速查看选项列表。这一点本身就已经足够提醒我们 Debugger 的选项所在的位置了。更好的方法是，找出完成同样任务的组合热键，缩短层层嵌套的菜单、对话框或调用顺序。

4.12 Debugger 快捷键

表 4-5 总结了最常用的 Debugger 选项及其相关的组合热键。

表 4-5 Debugger 组合热键

动作	键
恢复	F5
取消	ESC
单步进入	F11
单步跳过	F10
单步跳出	SHIFT+F11
运行到光标	CTRL+F10
Trace 窗口	ALT+8
Watch 窗口	ALT+3
Locals 窗口	ALT+4
调用堆栈窗口	ALT+7
Debug Output 窗口	ALT+2
开关断点	F9
清除断点	CTRL+SHIFT+F9
Breakpoints	CTRL+B
打开文件	CTRL+O
保存配置	ALT+S
退出 Debugger	ALT+F4

花一些时间熟悉该列表中的组合热键，将使得 Debugger 的使用更加有效和更加自如。

4.13 小结

本章提供了最常用的 Debugger 选项。在逐行执行一个算法时观察一个变量的内容，对于即时检测不正确的代码语句非常重要。

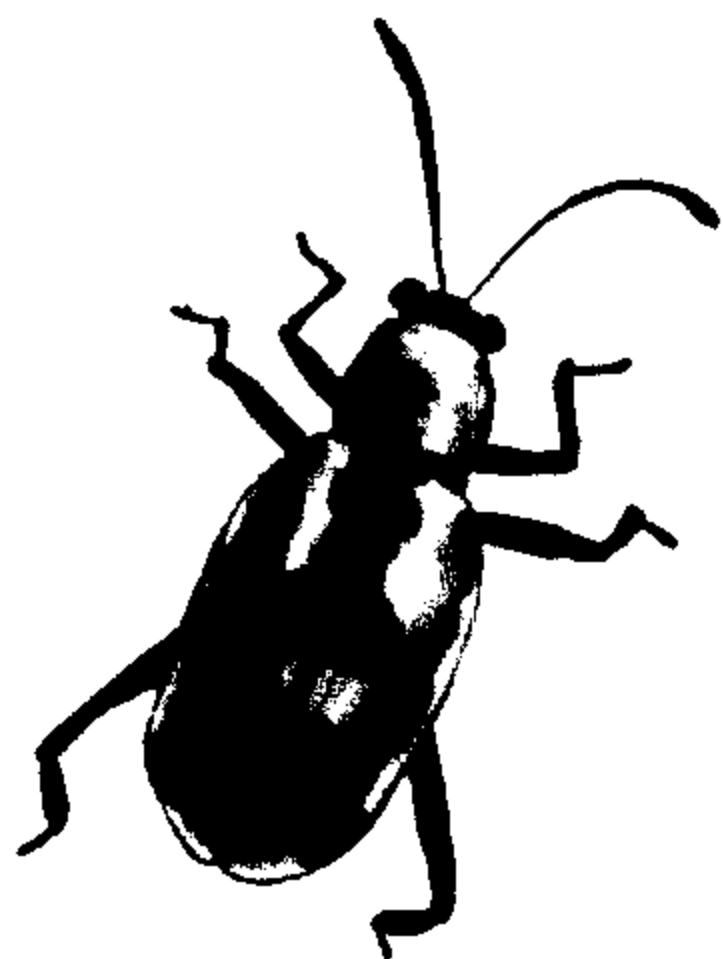
下一章中我们将讨论如何处理从调试版本改变为发行版本时，突然出现的新的程序错



误。最后，还将通过介绍所建立的不同版本观察源代码的不同，以及为什么在转换建立版本时可突然出现程序错误，并说明如何避免出现这种情况。 ■

第 5 章

调试版本与发行版本





本章将结束对 Microsoft Visual Studio C++ Debugger 的介绍。第 2 章“使用编译器优化”和第 4 章“Debugger 基础”提供了对各种 Debugger 功能及如何在 Visual Studio 开发环境下访问这些选项进行了深入讨论。本章中，我们将学习关于为调试和发行版本建立的附加代码因素，以及与之相关的 Visual C++ 库。在这些库中，随着双系统和远程调试的引入，有些变得非常重要。本章还将说明伴随着调试版本的建立而建立的各种中间文件。

严格地讲，Microsoft 将应用程序的调试版本定义为与符号调试信息，或与调试库一起建立的程序的一个版本。一个库的版本(例如，Microsoft Foundation Class 库)包括诊断帮助，并执行各种集成检查，以帮助调试一个程序。Microsoft 将发行版本定义为程序的一个编译版本，其中不包括在调试模式中建立编译时所包含的调试和诊断特性。例如，发行版本不包括 ASSERT 宏中所包含的源代码。

5.1 缺省的调试版本建立与发行版本建立设置

当开始一个新的 Visual C++ 工程时，将自动使用缺省选项建立 Win32 Debug 和 Win32 Release 版。表 5-1 中列出了这两种版本的初始缺省设置。

表 5-1 缺省调试(Debug)和发行(Release)版的编译器设置

建立版本	缺省编译器设置
Win32 Debug	这一选项以 Microsoft 格式，插入完整的符号调试信息，并且不做任何优化。在这一阶段使用优化选项将使调试更困难
Win32 Release	不包括任何符号调试信息。源代码以最快速度优化目标

在建立调试版本时，可以选择若干个选项而不影响到调试阶段。例如，可以改变缺省的 Debug Build 选项，如只输出行号，重定向输出或产生一个映射文件(mapfile)。

5.2 为调试版本建立修改工程设置

虽然可以重载许多缺省的 Debug Build 设置，但只有一些选项是安全的。表 5-2 列出了可以选择使用而不会严重影响由优化器执行的语句翻译顺序或类型的 Debug Build 选项。用户可以在 Project Settings 对话框(从 Project 菜单的 Settings 命令进入)的 C/C++ 标签中修改工程的设置。

表 5-2 安全的 Debug Build 选项

安全的 Debug Build 选项	导致 Debug Build...
Program Database for Edit and Continue(缺省)	产生一个附加输出文件称为程序数据库(.pdb)。该文件包含程序的类型信息和符号调试信息
Line Number Only	修改.obj 文件或可执行文件(.exe)的翻译，以使其只包含全局和外部符号以及行号信息(不包含符号调试信息)



(续表)

安全的 Debug Build 选项	导致 Debug Build...
C7 Compatible	产生一个.obj 文件和一个.exe 文件，并带有调试器使用的行号和全部的符号调试信息
None	不产生调试信息

5.2.1 修改调试选项

以下四步描述了修改 Debugger 选项的步骤：

1. 选择主 Project 菜单，然后单击 Settings，显示 Project Settings 对话框。
2. 单击 C/C++ 标签。
3. 从 Debug info 下拉列表选择一个选项。
4. 单击 OK 按钮，接受所做的选择，关闭 Project Settings 对话框。

5.2.2 修改产生调试信息的格式

以下五步详细描述了配置 Link 为 Microsoft Format(缺省)、COFF Format 或二者皆有的步骤：

1. 从 Project 菜单中单击 Settings，显示 Project Settings 对话框。
2. 单击 Link 标签。
3. 从 Category 下拉列表中选择 Debug 选项。
4. 选择 Microsoft Format(缺省)、COFF Format、或 Both Formats。
5. 单击 OK 按钮，接受所做的选择，关闭 Project Settings 对话框。

5.2.3 产生一个映射文件

在调试一个应用程序时，特别是当需要卸出特定的内存时，有时映射文件将非常有用。映射文件实际上是一个文本文件，其中包含了关于正在连接的程序的如下信息：

- 该文件的模块名称或基名称。
- 该程序头文件的时间戳。
- 程序组列表，每个组以 **segment:offset**(段：偏移量)形式的起始地址，以及长度、组名和类构成。
- 公共符号列表，各地址均以 **segment:offset** 形式给出，包括符号的名字、浮动地址、及定义符号的.obj 文件。
- **segment:offset** 形式的模块入口地址。

为打开映射文件的 Debug 选项，执行如下操作步骤：

1. 从 Project 菜单中单击 Settings，显示 Project Settings 对话框。



2. 选择 Link 标签。
3. 从 Category 下拉列表中选择 Debug 选项。
4. 选择 Generate mapfile 复选框。
5. 为所产生的输出文件在 Mapfile name 文本框中选择一个名字，或接受缺省名。
6. 单击 OK 按钮，激活设置后关闭 Project Settings 对话框。

5.2.4 重定向调试输入和输出

Microsoft Visual C++ 允许重定向任何 Win32 控制台应用程序的文件输入或输出位置，选择这些选项的步骤如下：

1. 单击 Project 菜单后选择 Settings 选项，显示 Project Settings 对话框。
2. 单击 Debug 标签。
3. 从 Program Arguments 文本框中指定一个或多个 I/O 重定向命令，这些命令在表 5-3 中给出。可以组合使用这些 I/O 重定向命令。
4. 单击 OK 按钮，激活重定向设置并关闭 Project Settings 对话框。

表 5-3 Win32 控制台应用程序的 Debug I/O 重定向命令

命令	动作
<filename	stdin 从 filename 输入
>filename	stdout 输出到 filename
>>filename	追加 stdout 到 filename
2>filename	stderr 输出到 filename
2>>filename	追加 stderr 到 filename
2>&1	发送 stderr(2)输出到与 stdout(1)相同的位置
1>&2	发送 stdout(1)输出到与 stderr(2)相同的位置

5.3 什么是.pdb 文件？

当 Debugger 打开了 Use Program Database 选项时，则在建立程序时将产生一个附加文件，即.pdb 或 Program Database 文件。该文件中包含了调试和工程的状态信息，允许 Visual Studio 执行调试程序版本的增量连接。

从历史的角度来看，Visual C++ 16 位版本的连接器将调试信息放置在.exe 或.dll 文件的尾部。现在，较新版本的 32 位的 Visual C++ 允许连接器和集成调试器直接使用.pdb 文件。

一般来说，Visual C++ 仅产生一个输出的.pdb 文件。然而，如果运行一个不是由 Microsoft Visual C++ 所创建的 makefile，则将发现有两个.pdb 文件。第一个文件为 VCx0.pdb(x 为当前 Visual C++ 的版本)，其中包含了所有针对独立的.obj 文件的调试信息。第二个文件为



projectname.pdb，其中包含了所有针对于 *projectname.exe* 文件的调试信息，可以在 \WINDEBUG 子目录下找到。

创建 VCx0.pdb 文件的原因是，编译器不知道将要产生的可执行文件的名称。在连接器激活之前，所需要的.obj 文件并未确定，所以这些信息暂时保存在文件 VCx0.pdb 中。每次编译一个.obj 文件时，编译器均将调试信息合并到 VCx0.pdb 中。

CVx0.pdb 和 *projectname.pdb* 文件中包含着不同类型的信息。VCx0.pdb 文件中所有关于程序类型的信息，但不包含任何符号信息。当应用程序包含一个标准的头文件如 windows.h 时，这种方法有一个很有效的优点。由于 VCx0.pdb 文件不包含符号信息，所以每一个.obj 文件都不带有重复 windows.h 符号信息的冗余开销。

projectname.pdb 文件是在连接阶段产生的。该文件包含了关于该工程的.exe 文件的调试信息。所有调试信息包括函数原型及其他信息，均存放在该 *projectname.pdb* 文件中。这两种类型的.pdb 文件具有相同的扩展名，其原因是它们的结构类似，二者均允许增量更新。Debugger 将在.exe 或.dll 文件中插入.pdb 文件的位置。如果路径改变，则 Debugger 可以在另一个位置定位该.pdb 文件。在这种情况下，Debugger 将使用.pdb 文件的当前缺省目录。

5.4 什么是.dbg 文件

最新的 Visual C++ 版本允许反汇编和调用堆栈窗口不修饰 C++ 的名字，在以前的版本中并非这样，例如当在 Windows NT .dbg(debug) 文件中显示系统调用堆栈时。现在，用户可以观察 C++ 的名字及其正确的变元列表。由 32 位的 NT 工具集所创建的.dbg 文件中包含 COFF 和 Codeview 两部分信息。Debugger 可以读取两种类型的.dbg 文件，然而却忽略 COFF 符号部分，只查看 Codeview 信息。

当没有源代码时，Debugger 仍可以使用该.dbg 文件，只要这些文件是使用包含 Codeview 格式的调试输出的二进制所建立。没有源代码时，我们仍可以使用.dbg 文件设置断点、观察变量和调用堆栈。

设计提示

由于优化代码中的信息已经重新排列，所以 Debugger 并不总是能够识别对应于一组指令的源代码。正因为如此，建议尽可能在优化之前调试代码。

5.5 调试优化的代码

从第 2 章中，我们知道优化可以导致编译器重新构造和/或重新排序源代码指令，以实现使用内存的最小化或执行性能的优化。当调试完成之后，可以打开优化选项。



从第 2 章中我们还知道,有些程序错误仅仅在优化的代码中出现,不影响未优化的代码。下面的讨论为必须调试优化代码的情况提供了一些建议。首先,打开 Program Database(可以使用 /Zi 编译器选项为应用程序获得最大可能的符号信息);其次,使用 Disassembly 和 Registers 窗口跟踪逻辑流和数据精度。这一阶段在 Disassembly 窗口中设置适当的断点将非常有用。

为了了解为什么 Disassembly 窗口会如此有用,考虑如下的 for 循环实例(加粗的代码语句强调了两种建立版本之间的一些区别):

```
for ( int i = 0; i < 5; i++ )
    cout << i;
```

第一个清单给出的是显示在 Disassembly 窗口中调试版本的翻译代码: (98 页)

```
5:      for ( int i = 0; i < 5; i++ )
00401778      mov     dword ptr [ebp-4],0
0040177F      jmp     main+2Ah (0040178a)
00401781      mov     eax,dword ptr [ebp-4]
00401784      add     eax,1
00401787      mov     dword ptr [ebp-4],eax
0040178A      cmp     dword ptr [ebp-4],t
0040178E      jge     main+40h (004017a0) // or OUT of loop
6:      std::cout << i;
00401790      mov     ecx,dword ptr [ebp-4]
00401793      push    ecx
00401794      mov     ecx,offset std::cout (004767e0)
00401799      call    @ILT+245 (std::basic_ostream
               <char,std::char_traits<char> >::operator<<)
               (004010fa)
0040179E      jmp     main+21h (00401781)
7:      }
```

第二个清单给出的是显示在 Disassembly 窗口中发行版本的翻译代码: (98 页)

```
00401080      push    esi
00401081      xor     esi,esi
5:      for ( int i = 0; i < 5; i++ )
6:      std::cout << i;
00401083      push    esi
00401084      mov     ecx,offset std::cout (00427318)
00401089      call    std::basic_ostream<char,std::char_traits<char>
               >::operator<< (004010a0)
0040108E      inc     esi
0040108F      cmp     esi,5
```



```
00401092    jl      main+3 (00401083) // back INTO loop!
00401094    pop     esi
7:    }
```

虽然两个清单中包含多处有趣的变化,但我们将仅集中讨论变量 *i* 的初始化。首先, C++ 程序在 **for** 循环中使用这一语句:

```
i = 0;
```

调试版本将该源代码语句翻译为一个等价的汇编语言指令:

```
00401778    mov     dword ptr [ebp-4], 0
```

然而, 发行版本使用等价的逻辑语句集, 将其优化成了更有效的代码(从一个机器循环的角度来看):

```
00401080    push    esi
00401081    xor     esi,esi
```

也可以比较 **for** 循环语句的最后两个部分: *i*<5 和 *i*++, 对其解释也是独特的。二者中较容易的是 LCV(循环控制变量)的增量 *i*, 在调试版本中 LCV 如下所示:

```
add     eax,1
```

在发行版本中使用了更有效的 **inc** 指令, 如下所示:

```
inc     esi
```

最后, **for** 循环的测试条件部分: *i*<5, 比较两种翻译似乎二者在逻辑上几乎完全相反。首先, 调试版本如下所示:

```
0040178A    cmp     dword ptr [ebp-4],5
0040178E    jge     main+40h (004017a0) // or OUT of loop
.
.
.
0040179E    jmp     main+21h (00401781)
```

注意, 从逻辑上看是预测试循环编码的部分, 在发行版本中类似于一个后测试循环:

```
.
.
.
0040108F    cmp     esi,5
00401092    jl      main+3 (00401083) // back INTO loop!
```



24x7

使用 `_DEBUG` 标签和条件预处理语句(指令), 允许有选择地调试代码段。当应用程序的大部分包含以前调试过的算法时, 考虑使用这种方法。

关键的问题是调试版本(使用了总共 12 条语句)和发行版本(使用了总共 9 条功能等价的语句)并非相同的孪生兄弟。在一个版本下运行正常的翻译单元在另一个版本下很容易发生故障。

5.6 打开 Debugger 的另一种方法

除了使用 Project|Settings 选项之外, 另外一种激活 Debugger 的方法是使用 `_DEBUG` 标签。这种方法允许我们有选择地打开和关闭 Debugger 开关。

下面的头文件语法演示了 `_DEBUG` 符号常量及 `#ifdef` 预处理指令的使用方法(正如在代码段中突出显示的那样, 一定要牢记每一个条件预处理指令 `#if...`, 必须有一个相应的结束语句 `#endif`): (100 页)

```
// Macros for setting or clearing bits in the CRT debug flag
#ifdef _DEBUG
#define SET_CRT_DEBUG_FIELD(a)  _CrtSetDbgFlag((a) |
    _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG))
#define CLEAR_CRT_DEBUG_FIELD(a) _CrtSetDbgFlag(~(a) &
    CrtSetDbgFlag(CRTDBG_REPORT_FLAG))
#else
#define SET_CRT_DEBUG_FIELD(a)  ((void) 0)
#define CLEAR_CRT_DEBUG_FIELD(a) ((void) 0)
#endif
```

在连接时必须使用 Microsoft Foundation Class(MFC)库的调试版本, 在 Visual C++中设置 Win32 Debug 选项可保证与调试版本的库连接。所有调试版本的库程序在其库名字的结尾处都有一个字母 D。静态调试版本的 MFC 是 `NAFXCWD.LIB`, 而静态发行版本(非调试版本)则是 `NAFXCW.LIB`, DLL 调试版本的 MFC 是 `MFCvD.LIB`, 而 DLL 发行版本(非调试版本)则是 `MFCv.LIB`(其中 `vv` 是版本号)。

5.7 使用基本版或调试版本

现在, Microsoft Visual C++运行时库包含了特殊的调试版本, 其中有与基本版中相同名字并增加了 `_dbg` 后缀的堆分配函数。本节将讨论调试版本与基本版在一个应用程序的调试版本建立中的区别, 其中所提供的信息以 `malloc` 和 `_malloc_dbg` 为例, 但适用于所有的堆分



配函数。

包含了现有的 malloc 调用的应用程序，为获得调试特性，无须将其调用转换为 _malloc_dbg。当 _DEBUG 定义后，所有对 malloc 的调用均将解释为 _malloc_dbg。然而，显式的 _malloc_dbg 调用允许对程序执行另外一些调试任务：可以单独跟踪 _CLIENT_BLOCK 类型的分配，可以在发生分配请求的保存在调试头中的记录信息中包含源文件和行号。

由于基本版的分配函数以包装形式实现，所以每一个堆分配请求的源文件名和行号不能通过显式调用基本版而获得。不需要将其 malloc 调用转换为 _malloc_dbg 的应用程序可通过定义 _CRTDBG_MAP_ALLOC 来获取源文件信息。定义 _CRTDBG_MAP_ALLOC 可使得预处理器直接将所有对 malloc 的调用映射为 _malloc_dbg，因此提供了附加信息。为了在客户块中单独跟踪特殊类型的分配，必须直接调用 _malloc_dbg，并且 blockType 参数必须设置为 _CLIENT_BLOCK。

当 _DEBUG 没有定义时：

- malloc 调用不受影响。
- _malloc_dbg 调用被解释为 malloc，_CRTDBG_BLOCK_ALLOC 定义被忽略。
- 不再提供关于分配请求的源文件信息。

由于 malloc 不存在块类型参数，所以对 _CLIENT_BLOCK 类型的请求将被处理为标准分配。

5.8 C/C++运行调试库

表 5-4、5-5 和 5-6 列出了调试版本的 Microsoft Visual C、C++ 及 iostream 运行调试库文件，与之相关的编译器选项和环境变量。在 Visual C++ 4.2 之前，C 运行库中包含 iostream 库函数。在 Visual C++ 4.2 中，早期的 iostream 库函数从 LIBCD.LIB、LIBCMTD.LIB 及 MSVCRTD.LIB 中删除。出现这一变化的原因是标准 C++ 库添加到了 Visual C++ 中，并且其中包含了一组新的 iostream 库。

表 5-4 C 运行调试库

C 运行调试库 (不包括 iostream)	描述	设置	使用
LIBCD.LIB	与静态连接单线程应用程序使用	/MLd	_DEBUG
LIBCMTD.LIB	与静态连接多线程应用程序使用	/MTd	_DEBUG, _MT
MSVCRTD.LIB (为 MSVCRTD.DLL 导入库)	与动态连接多线程应用程序使用	/MDd	_DEBUG, _MT, _DLL



(续表)

C 运行调试库 (不包括 iostream)	描述	设置	使用
用户可以找出应用程序正在使用的 DLL 版本, 查看文件名中 Visual C++ 的版本号。例如, 如果正在使用 Visual C++ 6.0, 则库的名字将是 MSVCR60D.DLL			

表 5-6 iostream 调试库

标准 C++ 调试库	描述	设置	使用
LIBCPD.LIB	对静态连接单线程的应用程序	/MLd	_DEBUG
LIBCPMTD.LIB	对静态连接多线程的应用程序	/MTd	_DEBUG, _MT
MSVCPRTD.LIB	对动态连接多线程的应用程序	/MDd	_DEBUG, _MT, _DLL

表 5-5 标准 C++ 调试库

iostream 调试库	描述	设置	使用
LIBCID.LIB	对静态连接单线程的应用程序	/MLd	_DEBUG
LIBCIMTD.LIB	对静态连接多线程的应用程序	/MTd	_DEBUG, _MT
MSVCIRTD.LIB	对动态连接多线程的应用程序	/MDd	_DEBUG, _MT, _DLL

错误监视

混合使用旧风格的 iostream.h、Standard Template Library(STL)及其相关的 using namespace std; 语句, 将导致数据丢失。相反, 一定要确保使用 iostream(不带.h)。

5.8.1 旧版 iostream.h 和新版 iostream 之间的混乱

现在, 两套 iostream 函数都包含在 Visual C++ 中, 这带来一些混乱。从某种程度上讲, 我们可以将 C/C++ 语言视为发展中的事物。“发展”的含义就是, 两种经 ANSI C/C++(American National Standards Committee)委员会通过的语言都在尽力满足当今编程开发的需要。

旧版的 iostream 函数现在存在于其自己的库中: LIBCID.LIB、LIBCIMTD.LIB 及 MSVCIRTD.LIB。新版的 iostream 函数以及其他许多新的函数存在于标准的 C++ 库中: LIBCPD.LIB、LIBCPMTD.LIB 及 MSVCPRTD.LIB。

标准 C++ 库和旧版 iostream 库并不兼容, 只有其中之一可以与我们的工程连接。当我们建立一个调试版本的工程时, 将缺省与基本的 C 运行调试库(LIBCD.LIB、LIBCMTD.LIB 或 MSVCRTD.LIB)之一连接。根据在代码中使用的头标的不同, 也会与来自标准 C++ 库或



来自旧版 `iostream` 库的一个调试库连接。

如果在代码中包含了一个标准 C++ 库头标，则在编译时 Visual C++ 将自动连接一个标准的 C++ 库。例如：

```
#include <iostream>
```

如果包含了一个旧版的 `iostream` 库头标，则在编译时 Visual C++ 将自动连接一个旧版的 `iostream` 库。例如：

```
#include <iostream.h>
```

注意，标准 C++ 库头标和旧版 `iostream` 库头标不能混合使用。头标决定了是连接一个标准 C++ 库，一个旧版 `iostream` 库，还是二者都不连接。编译器选项决定哪个库将作为缺省连接(单线程、多线程或 DLL)。当定义了一个特定库的编译器选项后，该库就被认为是缺省库，同时也自动定义了其环境变量。

切记，调试版本的 C/C++ 库函数与发行版本的库函数不同，特别是在使用 `/Z7` 或 `/Zi` 编译器选项编译 C/C++ 库函数，优化选项关闭，并且有源代码时，将包括调试信息。有些调试库函数中还包含着判断功能，用来验证参数的合法性。使用这些调试库中的任何一个都非常简单，如同使用 `/DEBUG:FULL` 连接器选项设置，将其与应用程序连接一样，然后即可直接执行几乎所有运行函数的调用。

5.9 连接器参考资料

Microsoft Visual C++ 的连接器是一个 32 位的工具，它可以将 Common Object File Format(COFF)目标文件与库连接，创建一个 32 位的可执行(.exe)文件或动态连接库(dll)文件。表 5-7 列出了许多 LINK.EXE 经常使用的选项。

表 5-7 Microsoft Visual C++ 连接器选项

连接器选项	描述
<code>/ALIGN: number</code>	选择各代码段的对齐方式
<code>/BASE: {address @filename.key}</code>	为程序定义基地址
<code>/COMMENT: ["comment"]</code>	在头中添加一个注释块
<code>/DEBUG</code>	插入调试信息
<code>/DEBUGTYPE: CV</code>	选择 CV 调试类型
<code>/DEBUGTYPE: COFF</code>	选择 COFF 调试类型
<code>/DEBUGTYPE: BOTH</code>	包括两种类型的调试格式
<code>/DEF: filename</code>	发送模块定义或 .def 文件到连接器
<code>/DEFAULTLIB: library</code>	在外部库引用搜索路径中使用所定义的文件
<code>/DELAY [NOBIND 或 UNLOAD]</code>	延迟加载 dll
<code>/DELAYLOAD</code>	定义使用延迟加载的文件



(续表)

连接器选项	描述
/DLL	创建一个 DLL
/DRIVER[:UPONLY]	建立一个 Windows NT 内核模式的驱动程序
/ENTRY:function	设置启动地址
/EXETYPE:DYNAMIC	创建一个虚拟设备驱动程序
/EXPORT	导出一个函数
/FIXED[:NO]	建立一个仅在指定的基地址加载的应用程序
/FORCE[:{MULTIPLE UNRESOLVED}]	使连接器忽略未定义或重复定义的标识符
/GPSize:#	为 MIPS 和 Alpha 结构设置公共变量大小
/HEAP:reserve[,commit]	以字节为单位限定堆的大小
/IMPLIB:filename	允许选择另一个导入库
/INCLUDE:symbol	强制符号引用
/INCREMENTAL:{YES NO}	确定增量连接
/LARGEADDRESSAWARE	允许一个应用程序具有大于 2Gb 的地址
/LIBPATH:path	选择另一个环境库路径
/LINK50COMPAT	以 Visual C++ 5.0 格式建立一个导入库
/MACHINE:{IX86 ALPHA ARM MIPS MIPS16 MIPS R41XX PPC SH3 SH4}	选择目标结构
/MAP	产生一个映射文件
/MAPINFO:{EXPORTS FIXUPS LINES}	添加所选择的信息到映射文件
/MERGE:from=to	合并指定的节
/NODEFAULTLIB[:library]	当处理外部引用时忽略所选择的缺省库
/NOENTRY	产生一个仅有资源的 DLL
/NOLOGO	取消启动标记
/OPT:{REF NOREF NOWIN98 WIN98ICF[,iterations] NOICF}	选择连接器优化选项
/ORDER:@filename	以预先确定的格式将 COMDAT 插入到影像中
/OUT:filename	选择一个输出文件
/PDB:filename	产生 Program Database, .pdb 文件
/PDBTYPE:{con[solidate] sept[ypes]}	选择 Program Database 调试类型信息的位置
/PROFILE	打开描述文件
/RELEASE	确定.exe 头的校验和
/SECTION:name,[E][R][W][S][D][K][L][P][X]	重载一个节的属性
/STACK:reserve[,commit]	以字节为单位定义堆栈大小
/STUB:filename	将一个 MS-DOS 应用程序附加到一个 Win32 应用程序
/SUBSYSTEM:{CONSOLE WINDOWS NATIVE POSIX WINDOWSCE}{,major[,minor]}	指定操作系统运行.exe 文件的方式
/SWAPRUN:{NET CD TSAWARE[:NO]}	指示操作系统在执行之前复制连接器输出到一个交换文件
/VERBOSE[:LIB]	输出连接器进展消息
/VERSION:major[,minor]	插入版本号



(续表)

连接器选项	描述
/VXD	产生虚拟设备驱动程序 VxD
/WARN[:level]	选择警告级别
/WS:AGGRESSIVE	优化内存使用

5.10 在调试版本中检测发行版本错误

有些程序错误只有在转换到发行版本(/O1、/O2、/Ox 或/Og)时才出现。我们可以使用/GZ 选项，打开运行检查，以在调试版本(/Od)中捕获这些程序错误。/GZ 与/O1、/O2、/Ox 或/Og 不兼容(/GZ 选项将关闭代码中的所有#pragma 优化语句)。当使用/GZ 选项时，Visual C++将自动初始化所有局部变量，检查函数指针调用堆栈的合法性，并检查调用堆栈的合法性(此处所讨论的许多例子均将在以后各章中作为例子使用)。

5.10.1 局部变量的自动初始化

假设堆栈为 0，则包含有未初始化变量的代码使用/GZ 时可能失败。考虑如下的例子。当使用/Od 或/Ox 而不使用/GZ 编译该代码时，将产生一个非法访问异常，也可能看上去运行正常。当使用/GZ /Od 编译时，则总是出现该异常。可以在 Debugger 中捕获该异常，并获得关于该故障确切位置的某些信息。

```
#include <stdio.h>
void myFuncA(char **ppszString)
{
    /* output uninitialized ppszString address in hex */
    printf("*ppszString = 0x%X\n", *ppszString);
    if(*ppszString == NULL)
        *ppszString = "/GZ option ACTIVE";
}
void myFuncB()
{
    int memBuff[25]; /* uninitiailed internal variable */
    char * pszString; /* uninitialized memBuff pointer */
    myFuncA(&pszString);
    puts(pszString); /* dangerous - attempts to print */
} /* myFuncA allocated string!!! */
void main( void )
{
    int memBuff[ 1000 ]; /* stack page - cleared */
}
```



```
myFuncB( );  
}
```

通过调试该应用程序，可以看出未初始化的 *pszString* 指针在使用或不使用/GZ 开关时的具体行为。当使用该开关时，该指针自动初始化为 NULL。在两种情况下，*myFuncB()* 函数的最后一条语句都试图执行一个危险的输出——访问一个被调用子程序的内部变量。

5.10.2 检查函数指针调用堆栈的合法性

函数指针调用堆栈合法性检查确保对堆栈指针在通过函数指针调用之前和调用之后检查，以确保两种情况下函数指针一样。当使用函数指针调用时，可以捕获在调用函数的清除调用规范 `__cdecl` 与被调用函数的清除调用规范 `__fastcall` 和 `__stdcall` 之间的错误。

本例使用/Od 时运行正常，而使用/Ox 时运行失败，使用/GZ 时产生一个异常。如果使用一个发行版本的运行库编译，则将得到一个异常断点弹出，当使用调试版本运行库编译时，可以得到一条更有意义的消息。

```
void __stdcall myFuncA(char *p)  
{  
    puts(p);  
}  
typedef void (*function_pointer)(char *);  
void main( void )  
{  
    function_pointer pmyFuncA = (function_pointer)myFuncA;  
    pmyFuncA("This is just a test.");  
}
```

本例演示了使用调用规范修饰符时可能遇到的各种问题，并演示了 Debugger 根据所产生的版本是调试版本或发行版本，标志不同错误的方式。

5.10.3 检查调用堆栈的合法性

当处于活动状态时，Microsoft Visual C++ 在函数的末尾处检查堆栈指针，查看堆栈指针是否被修改。这可以捕获这样一些错误：堆栈指针在内联汇编中遭到破坏，一个非指针函数的调用规范没有正确设置。

5.11 TRACE 宏

Debugger 的 TRACE 宏可显示一个程序的调试消息，这些消息输出到 Debugger。



说明

对于 32 位的 MFC，获得调试输出信息的唯一方法是通过 Debugger。

与 printf() 函数类似，TRACE 宏可以处理任意多个参数。下面是以不同方式使用 TRACE 宏的例子：

```
char a = 'a';
char b = 'b';
TRACE( "TRACE statement." );
TRACE( "\na = %c", a );
TRACE( "\nb = %c", b );
```

TRACE 宏仅在调试版本的类库中有效。当已经调试一个程序后，可以建立一个发行版本解除程序中所有的 TRACE 调用。

提示

当调试 Unicode 时，TRACE0、TRACE1、TRACE2 和 TRACE3 宏更容易使用，因为此时不再需要 _T 宏。

5.12 VERIFY 宏

在建立一个 MFC 调试版本时，可以使用 VERIFY 宏对参数求值。当 VERIFY 返回 0 时，该宏打印一条诊断消息并挂起程序，如果返回非 0 值，则不执行任何操作。诊断消息的语法如下所示：

```
assertion failed in file <identifier> in line <ln>
```

其中，*identifier* 为源文件的名字，*ln* 为源文件中出现错误的行号。

当建立一个 MFC 发行版本时，可以使用 VERIFY 宏求表达式的值，而不打印消息或中断程序。例如，如果一个表达式是一个函数调用，则将正常执行调用。

5.13 移植 Visual C++ 旧的 32 位版本

幸运的是，所有在 Microsoft Visual C++ 以前的 32 位版本下建立的早期的应用程序工作空间或工程，均可以自动移植到新的 Visual C++ 环境下。但是，如果使用自己的 makefile，从命令行建立了一个 32 位的 Visual C++ 工程，则 Visual C++ 开发环境将不能识别文件中的工程数据。相反，开发环境提供了对该 makefile 的包装。如果选择这一选项，将创建一个包含该 makefile 的工程文件，但该工程将仍在开发环境中建立和运行。



说明

然而, 如果需要使用新的 Microsoft Visual C++ 向导和集成调试特性, 则应该在开发环境中创建一个新的工程, 添加现有的文件, 然后建立一个更新的工程后保存该工程。

5.13.1 转换早期的 32 位工作空间和工程

转换早期的 Visual C++ 32 位工程只需要执行如下 4 个简单步骤即可。首先, 单击 File|Open Workspace 菜单选项。当 Open Workspace 对话框出现后, 执行如下步骤:

1. 在 Files of Type 下拉列表中选择 All Files(*.*)。
2. 在磁盘和目录中查找以前的 makefile 或工程文件。
3. 双击文件列表中的 makefile 或工程文件。
4. 当出现消息框询问是否需要转换为新的工程格式时, 单击 Yes。

Microsoft Visual C++ 创建一个新的工程工作空间文件(.dsw), 并使用原来的工程工作空间名创建一个新的工程选项文件(.opt)。另外, 在该工作空间中为每一个工程在各目录下建立一个工程建立文件(.dsp)。例如, 假设我们正在转换一个只有一个工程的工作空间, 且命名为 oldapptoconvert.mdp, 那么将建立文件 oldapptoconvert.dsw、oldapptoconvert.dsp 及 oldapptoconvert.opt, 并将其保存在相同的目录下。在一个更复杂的例子中, 工作空间中包含了两个工程, 名为 oldapptoconvert2.mdp, 其中还包含一个子工程 myOwndll2, 那么将建立文件 oldapptoconvert2.dsw、oldapptoconvert2.dsp 以及 oldapptoconvert2.opt, 并保存在同一目录下, 而文件 myOwndll2.dsp 则建立并保存在 myOwndll2 目录下。

5.13.2 与 Visual C++ 以前的版本共存

对于各种新版本的 Microsoft Visual C++ 开发环境, 所有的新安装程序均将覆盖以前的 Visual C++ 版本所使用的早期的库 DLL 和 PDB。Visual C++ 4.2 和 5.0 不能读取这些 DLL 中新的调试信息格式(表 5-8 列出了 6.0 与 4.2 和 5.0 有影响的文件)。当需要将 Visual C++ 6.0 与 4.2 或 5.0 一起使用时, 可以按照如下方法处理:

- 为每一个环境使用单独的机器。
- 在同一台机器上使用不同的 Visual C++ 版本与各自独立的操作系统安装。
- 使用 Visual C++ 6.0 Debugger 在新的 DLL 中调试。
- 复制受影响文件的老版本到 Visual C++ 5.0 或 4.2 工程的 Debug 目录中(在安装 Visual C++ 6.0, 或从 Visual C++ 4.2 或 5.0 CD 的 \MSDEV\DEBUG 目录获得早期版本之前, 完成这一复制)。



表 5-8 当混合使用 Microsoft Visual C++ 版本时受影响的文件

受影响的 CRT 文件	MSVCRTd.dll、MSVCRTd.pdb、MSVCIRTD.dll、MSVCIRTD.pdb
受影响的 MFC 文件	MFC42d.dll、MFC42d.pdb、MFC42u.pdb、MFC42ud.dll、MFC42ud.pdb、 MFCd42d.dll、MFCd42d.pdb、MFCd42ud.dll、MFCd42ud.pdb、MFCn42d.dll、 MFCn42d.pdb、MFCn42ud.dll、MFCn42ud.pdb、MFCo42d.dll、MFCo42d.pdb、 MFCo42ud.dll、MFCo42ud.pdb

Microsoft 建议不要使用早期版本替换新版的 MFC42.dll、MFC42u.dll 及 MSVCRT.dll。如若不然，根据新版本的这些 DLL 的不同，将损坏其他的应用程序。例如，Visual C++ 6.0 开发环境要求有当前版本的 MFC42.dll。如果用户需要使用 Visual C++ 6.0 以 Visual C++ 5.0 格式产生导出库，则可以使用 /LINK50COMPAT 选项。

5.14 小结

读者即使已经熟悉了标准 Debugger 的基本知识，如 Watch 窗口、Step Into、Step Over，也可能并不知道 Debugger 执行复杂任务的功能，如从当前位置使用新值调试。本章还提供了一些高级主题，如调用堆栈和通过灵活地插入断点有效使用标准 Debugger 的基本功能。随着以后的实践，读者将越来越有效地调试，并花费越来越少的时间调试“早期的问题”。

下一章中，我们将学习 Debugger 与汇编语言代码相关的功能。这是程序员工具箱中一个激动人心的附件，因为从那时起，C/C++ 就与汇编语言代码有了同甘苦共命运的关系。



第二部分

DEBUGGING

DEBUGGING

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

DEBUGGING

面向过程的环境

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

- 第 6 章 定位、分析和修复命令行代码错误
- 第 7 章 调试内联汇编语言代码
- 第 8 章 在 Windows 代码中定位、分析和修复错误

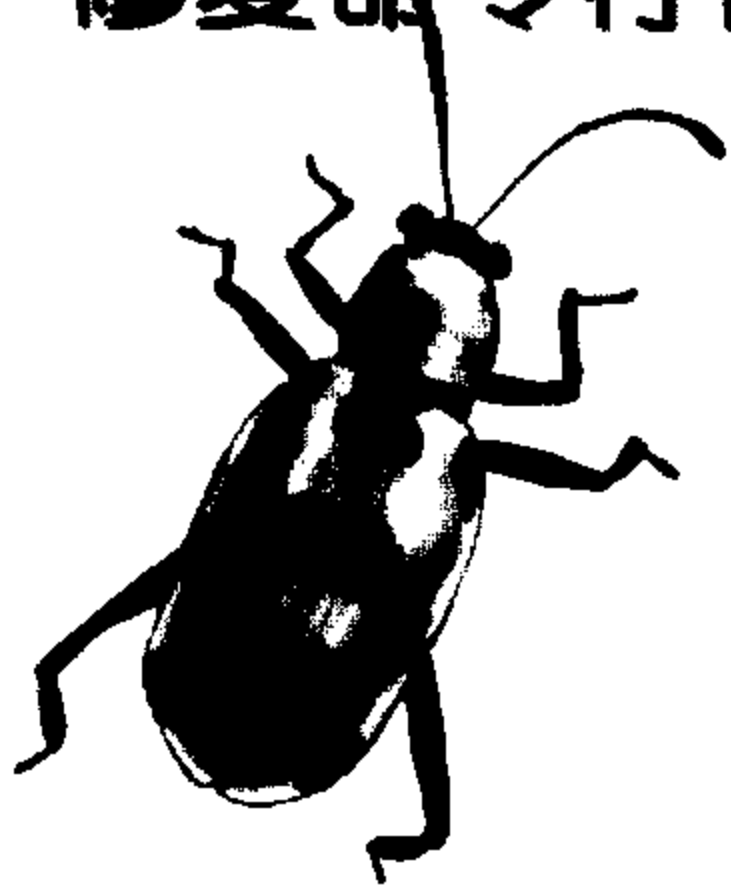
DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING



DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

第 6 章

定位、分析和
修复命令行代码错误





现在，有许多类型的 C/C++ 开发环境和体系结构可供程序员使用。在第 6 章中我们有两个主要目的：第一，为不熟悉 Microsoft C++ Debugger 的程序员介绍一部分有意义的 Debugger 命令，使其立刻即可使用这些命令；第二，将这些命令应用于 Microsoft 所谓的“Win32 Console Application”（控制台应用程序）。

在第 6 章的后半部分“高级 Debugger 技巧”中将介绍一些调试的基本原理。这些基本原理与添加到一个或多个 Debugger 命令的组件结合，可以有效地定位和诊断程序错误。

6.1 快速启动调试

此处，我们循序渐进地介绍 Visual C++ Debugger，假设读者从未使用过这一产品。每一个连续的图都详细而突出地说明了应该在监视器屏幕上看到的确切样子。由于 Debugger 是一个与程序员的交互式产品，所以建议读者输入、编译、建立这些例子程序，然后试运行每一个 Debugger 命令选项。

另外，读者可能需要在笔记本上对 Debugger 命令顺序做一些笔记。本章的讨论中经常详细介绍多种鼠标/菜单、组合键/热键，供读者访问某一个 Debugger 特性，读者可能有自己的喜好。虽然 Microsoft 的 Visual C++ Debugger 具有外科诊断能力，但我们并不是每天都要使用数百条 Debugger 命令。相反，我们只使用一个很小的子集。记录这些命令的名字和程序员/产品交互顺序，并不占用很多时间，但在随后的调试中将节省时间立即得到回报。

24x7

ANSI C++ Committee(委员会)已经修改了 C/C++ 头文件使用的语法、范围及内部表示。本书中的所有程序都采用了新标准。读者可以通过检查 `#include` 语句，识别一个最新式的算法。一个使用以 `.h` 结束的标准 C/C++ 头文件名的程序不是最新的程序(ANSI C++ 的标准有很多内容，但不适合在此处讨论)。另外，最新 ANSI C++ 的改进，不仅头文件不再使用 `.h` 扩展名，而且要同时使用一个“`using namespace std;`”语句，以编译并运行该应用程序。

现在，如果准备试运行本章中的例子，则需要输入并保存下面的 `VarsYourCode.cpp` 程序：

```
//  
//VarsYourCode.cpp  
//Debugging YOUR code versus Stroustrup's  
//  
#include <iostream>  
#include <string>  
using namespace std;  
void main ( void )
```



```
{
    //sample variables of common data types
    char cValue = 'A';
    int iValue = 10;
    float fValue = 3.19143;
    string szString = "Oh NO - not another Hello World!";
    cin >> szString;
    cout << szString << endl;
}
```

图 6-1 给出了一个简单地编辑和保存的样本程序 VarsYourCode.cpp。此时，还没有执行任何编译和建立指令。

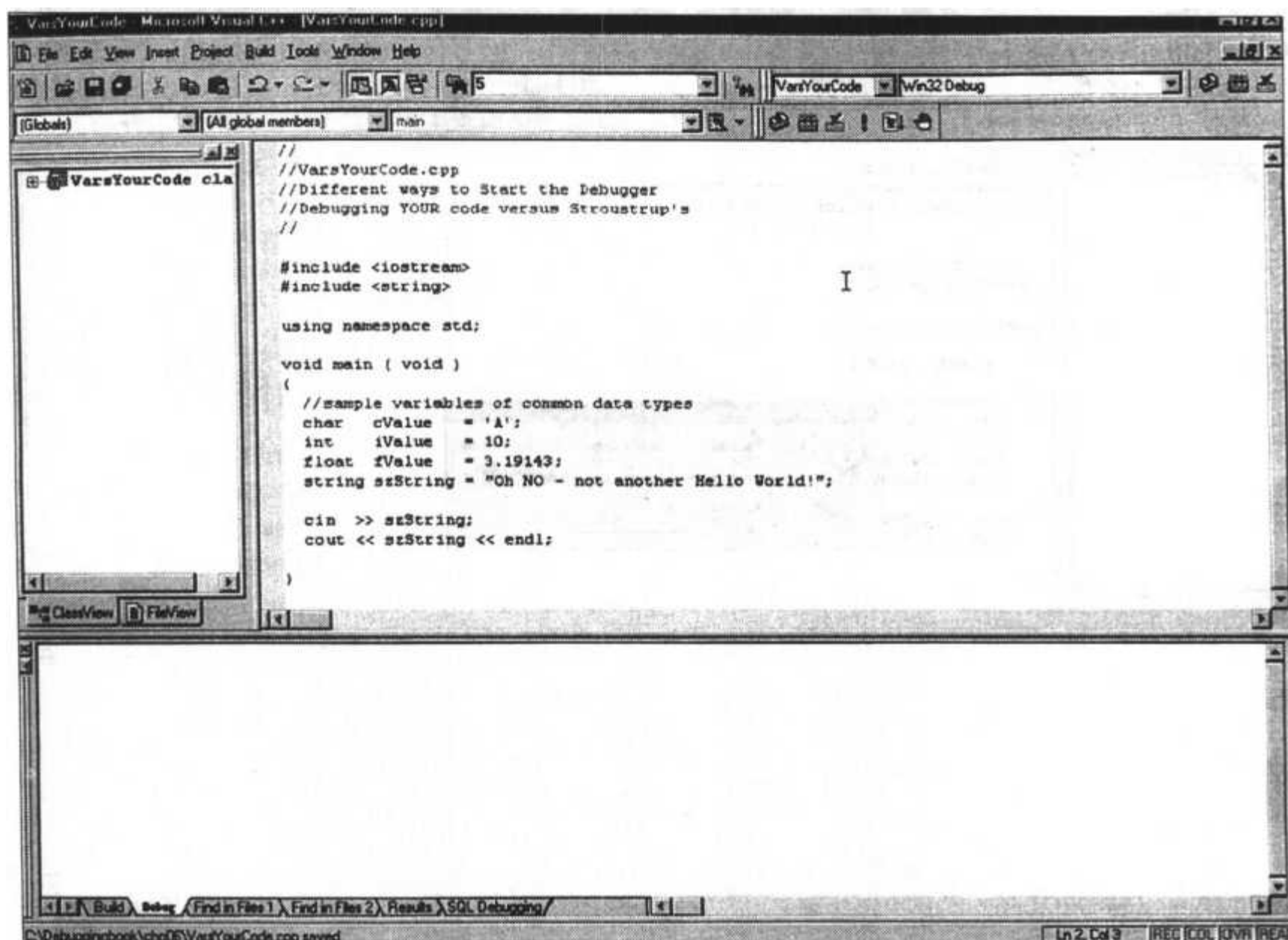
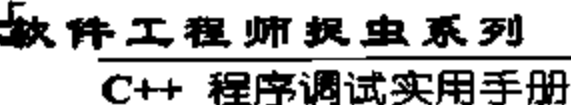


图 6-1 在建立、编译或调试之前的样本程序 VarsYourCode.cpp

说明

本章中任何需要仿效的动作实例都采用倾斜加粗并带有下划线字体。讨论中的有些部



6.1.1 启动 Debugger 的快速方法

6.1.1.1 第一步——按 F10

The screenshot shows the Microsoft Visual C++ 6.0 IDE. The main window displays a C++ program named `VarsYourCode.cpp`. The code includes headers for `<iostream>` and `<string>`, uses the `std` namespace, and defines a `main` function. The `main` function contains sample variable declarations and a loop that reads and prints strings.

A debug console window is open, showing the output of the program. The output consists of the strings "1234567890" and "0123456789" printed on separate lines. The status bar at the bottom indicates the program is in the "Debug" state, and the "Find in Files" dialog is open.

```
//
//VarsYourCode.cpp
//Different ways to Start the Debugger
//Debugging YOUR code versus Stroustrup's
//

#include <iostream>
#include <string>

using namespace std;

void main ( void )
{
    //sample variables
    char  cValue;
    int   iValue;
    float fValue;
    string szStr;

    cin >> szStr;
    cout << szStr;
}
```

```
1234567890
0123456789
```

新的消息窗口提示用户将要创建该程序的一个可执行版本 VarsYourCode.exe。注意，选择 Yes(或按 ENTER 键)不仅启动了创建.exe 版本的过程，而且自动启动了一个编译/建立的

过程。

6.1.1.2 第二步——单击 Yes

只要单击 **Yes** 按钮(或者只按 ENTER 键更好, 因为 Yes 按钮是缺省操作)即可。图 6-3 可能由于缺省工具栏位置的不同与读者自己的屏幕略有区别, 但窗口的内容应该是相同的。

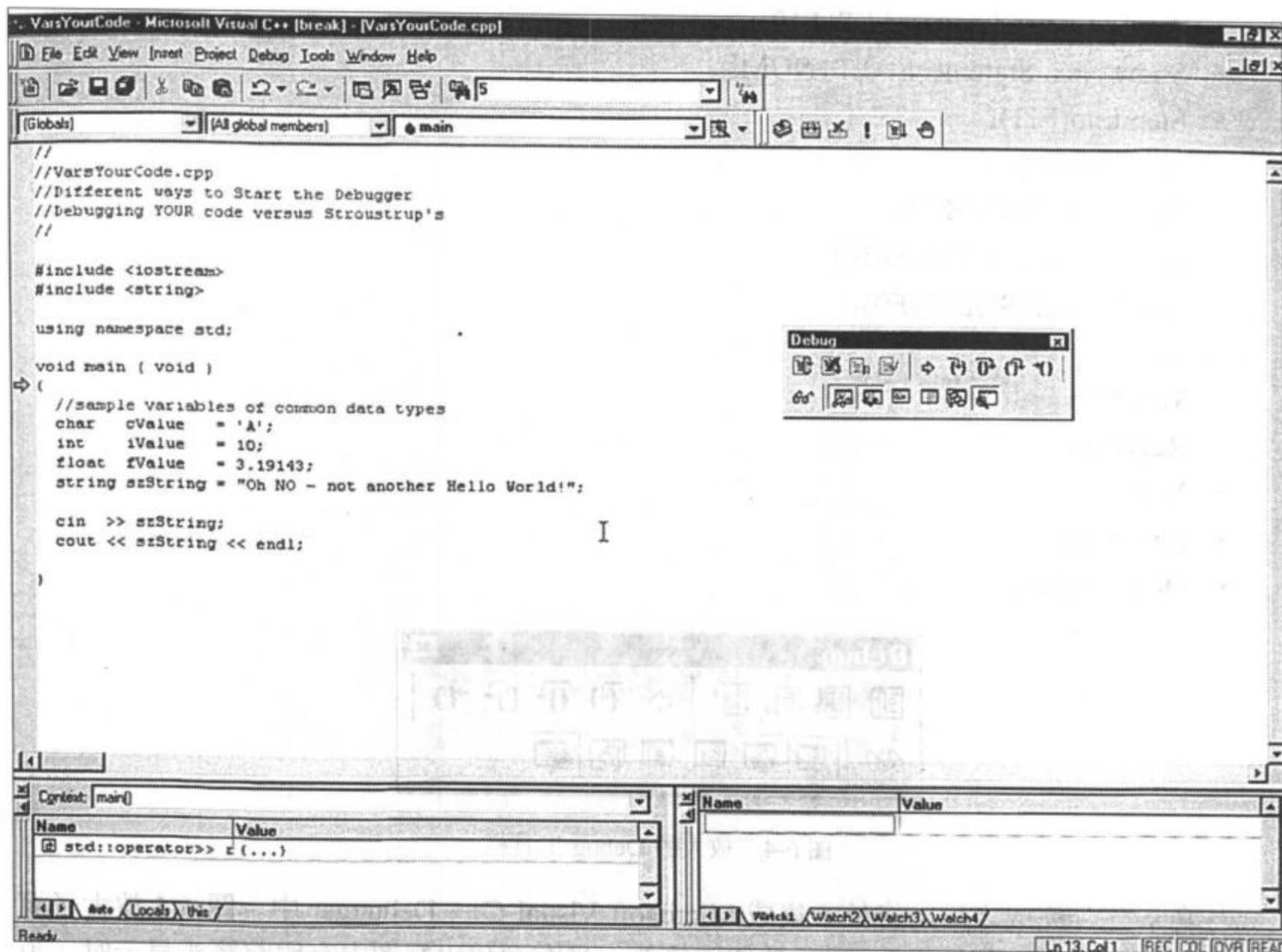


图 6-3 单击“Do you want to build it?”消息框中的 Yes 按钮后的结果

图 6-3 给出了更新以后的 Visual C++ 窗口内容。首先, 注意 Debug 工具栏。如果读者在自己的系统上调试该实例, 那么工具栏可能处于不同的位置, 也可能有不同的形状, 另外还可能在屏幕上源代码的左边显示 Workspace View 窗格。在这一例子中, 隐藏了 Workspace View 窗格, 以尽可能减少屏幕上的混乱, 使读者集中精力于正在讨论的问题(右击该窗格后选择 Hide 可隐藏该窗格)。

图 6-4 给出了放大的 Debug 工具栏, 其中的 16 个图标自左至右、自第一行到第二行列



表说明如下, 其中还给出了相关的组合热键(第 4 章中详细描述了每一个 Debug 工具栏按钮的操作):

- Restart(CTRL+SHIFT+F5)。
- Stop Debugging(SHIFT+F5)。
- Break Execution(暗淡显示)。
- Apply Code Changes(ALT+F10)。
- Show Next Statement(ALT+NUM*)。
- Step Into(F11)。
- Step Over(F10)。
- Step Out(SHIFT+F11)。
- Run to Cursor(CTRL+F10)。
- QuickWatch(SHIFT+F9)。
- Watch。
- Variables。
- Registers。
- Memory。
- Call Stack。
- Disassembly。

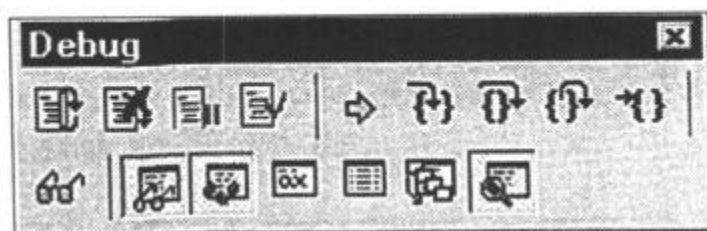


图 6-4 放大的 Debug 工具栏

当编译/建立完成之后, 将处于集成 Microsoft Visual C++ Debugger 中。图 6-5 放大了源代码窗格, 以使读者看清 Debugger 的跟踪箭头。当在源代码视图中有跟踪箭头显示时, 说明 Debugger 正在运行。

```
Void main {void}
{
    //sample variables of common data types
    char cValue = 'A'
```

图 6-5 放大的跟踪箭头

关于跟踪箭头应记住的最重要的一点是, 箭头所指的代码语句不是刚执行过的语句, 而是下一次执行 Single Step Into/Over(F11 或 F10)时, 将要执行的代码行。



在 VarsYourCode.cpp 例子中,跟踪箭头停放在起始处的花括号“{”处,正好处于 `main()` 的下面。停放在此处使得我们可以观察到所有变量的初始化代码。跟踪箭头并不跳过变量的声明部分而直接停放在第一条“可执行的”语句代码处。

图 6-6 给出了放大显示的自动跟踪信息 Variables 窗口,该窗口自动跟踪关于在当前跟踪箭头的范围内所使用的变量信息。Variables 窗口自动地更新其内容,使其成为当前语句和前面所执行过的语句的变量使用情况和函数返回值(在 Auto 标签中),关于当前函数的局部变量(在 Locals 标签中),以及由 `this` 所指向的对象(在 This 标签中)。Variables 窗口的 Context 下拉列表指出了“拥有”当前环境的子程序。

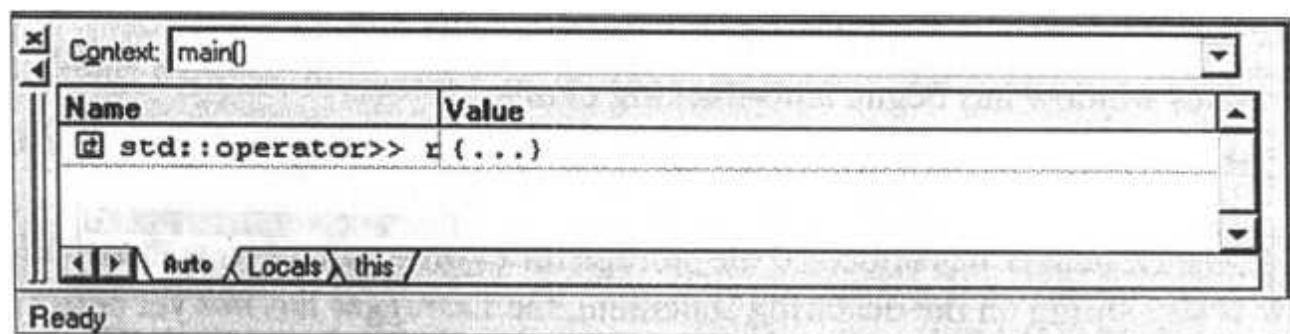


图 6-6 放大显示的 Variables 窗口

图 6-7 显示的是 Watch 窗口,当前该窗口为空。该窗口的内容与 Variables 窗口不同,其中所插入的标识符不超出范围,Variables 窗口的内容对跟踪箭头的范围自动跟踪或同步。

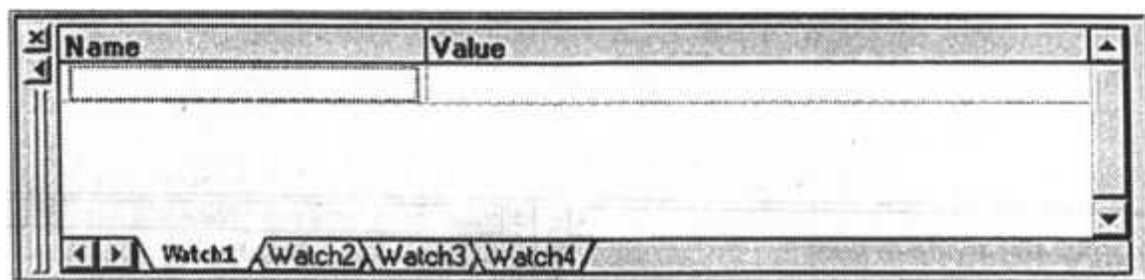


图 6-7 Debugger 的 Watch 窗口

6.1.2 变量初始化跟踪

下面将解释 Debugger 跟踪变量声明和初始化时的操作。如果读者正在调试所给出的实例,则可以(但现在不要执行这些所推荐的选项):

- 按 F10 键。
- 再按 F11 键。
- 单击 Debug 工具栏的 Step Into 按钮。
- 单击 Debug 工具栏的 Step Over 按钮。



用户可以选择四种选择中的任何一种，原因是对于当前跟踪箭头所指语句的语法来说，四种方法所实现的功能完全相同。切记，Step Over 和 Step Into 的功能是相同的，除非跟踪箭头处于一个子程序(函数或方法)调用上。

有些程序员更喜欢使用组合热键，而有些则更喜欢简单的单击。读者可根据自己的喜好，选择上述四种方法之一，此时将看到类似于图 6-8 所示的屏幕。

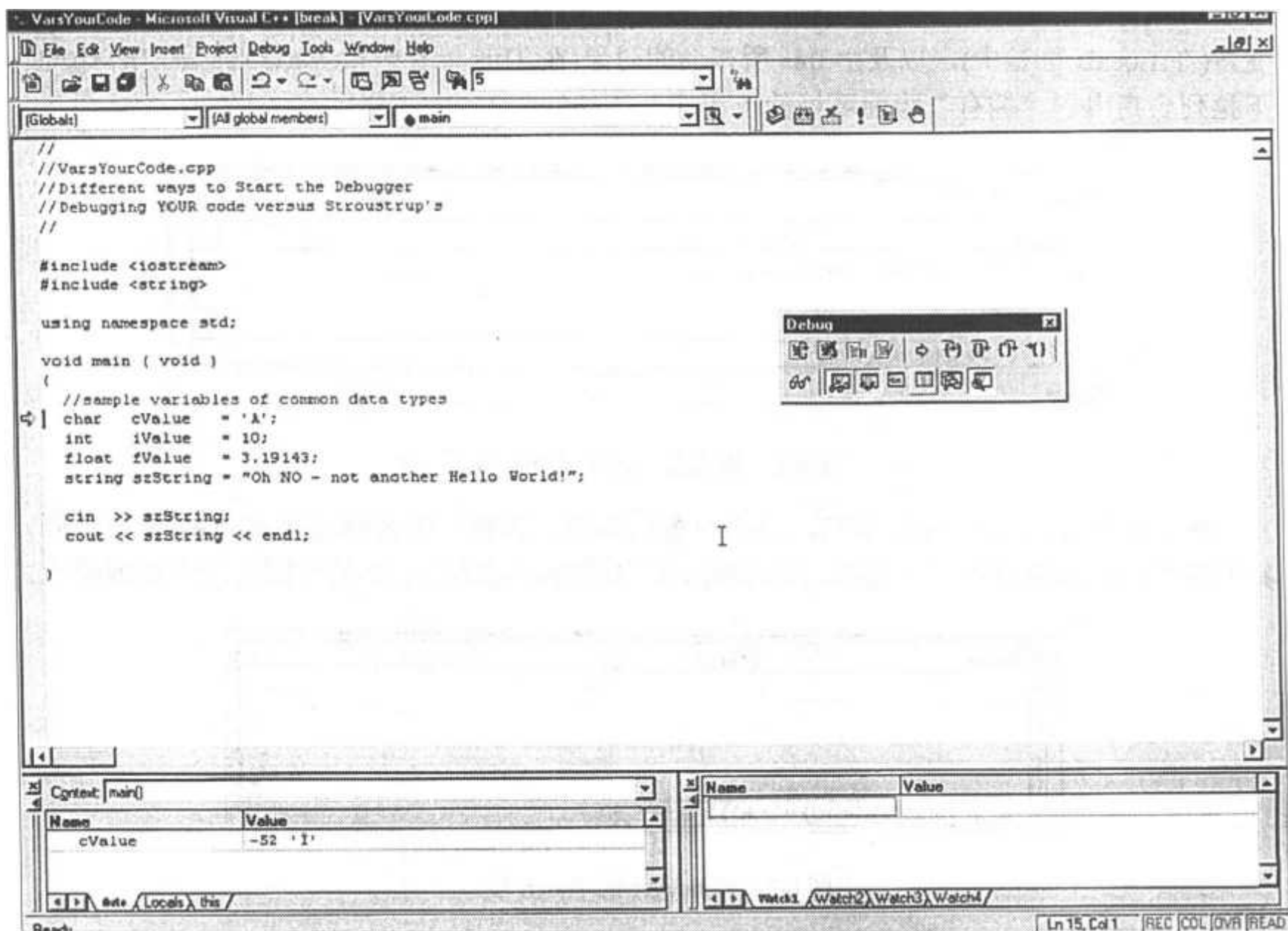


图 6-8 执行了一个单步跟踪后的窗口内容

应该可以看到 Visual C++窗口的如下一些变化：

- 跟踪箭头现在移动到了 cValue 声明行。
- Variables 窗口已经开始自动跟踪 cValue。
- Variables 窗口的 Value 列显示的 cValue 的初始内容为垃圾。

此时，Debugger 已经为 cValue 分配了存储空间。但正如我们已经知道的那样，由于跟踪箭头仍然停留在声明语句上，Debugger 还没有执行初始化。

现在，使用自己所喜欢的方法，再执行一步。新的 Visual C++ 窗口内容应如图 6-9 所示。应该可以看到 Visual C++ 窗口有如下一些变化：

- 跟踪箭头在代码中向下移动了一行。
- Variables 窗口现在显示了正确的 cValue 初始化值。
- Variables 窗口开始自动跟踪其范围内的变量 iValue。
- Variables 窗口 Value 列显示的 iValue 初始内容为垃圾。

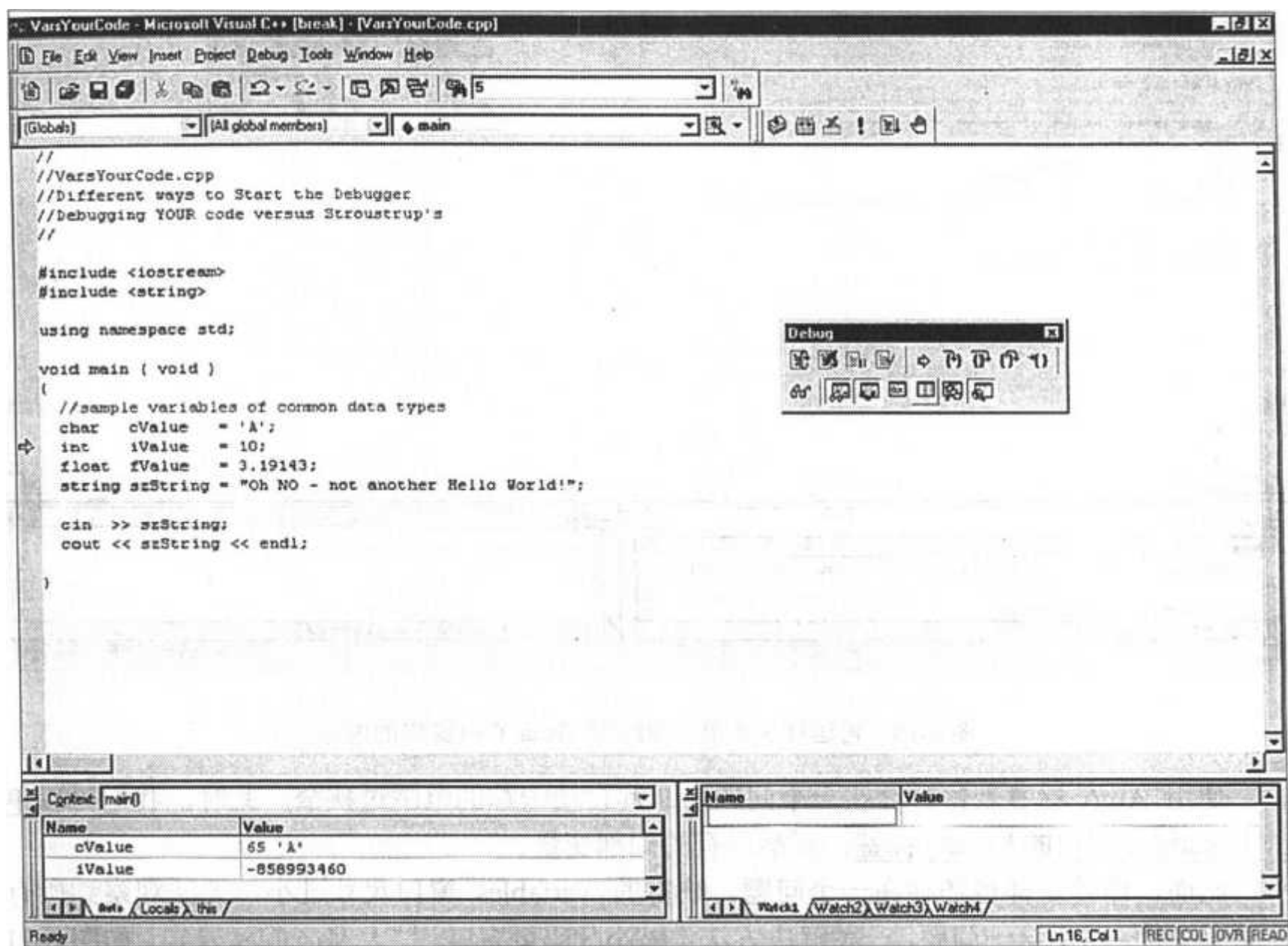


图 6-9 执行 cValue 的初始化

此时，需要再执行 3 个单步执行。请特别注意 Variables 窗口中的动态变化。这种情况将重复出现：

- 首先看到最新添加的变量的内容从一个垃圾值变为一个显式代码指定的值。
 - 接着可注意到下一个变量名出现在 Variables 窗口中，最新变量的内容显示为垃圾。
- 当单步跟踪完变量声明块后，Visual C++ 窗口的内容如图 6-10 所示。

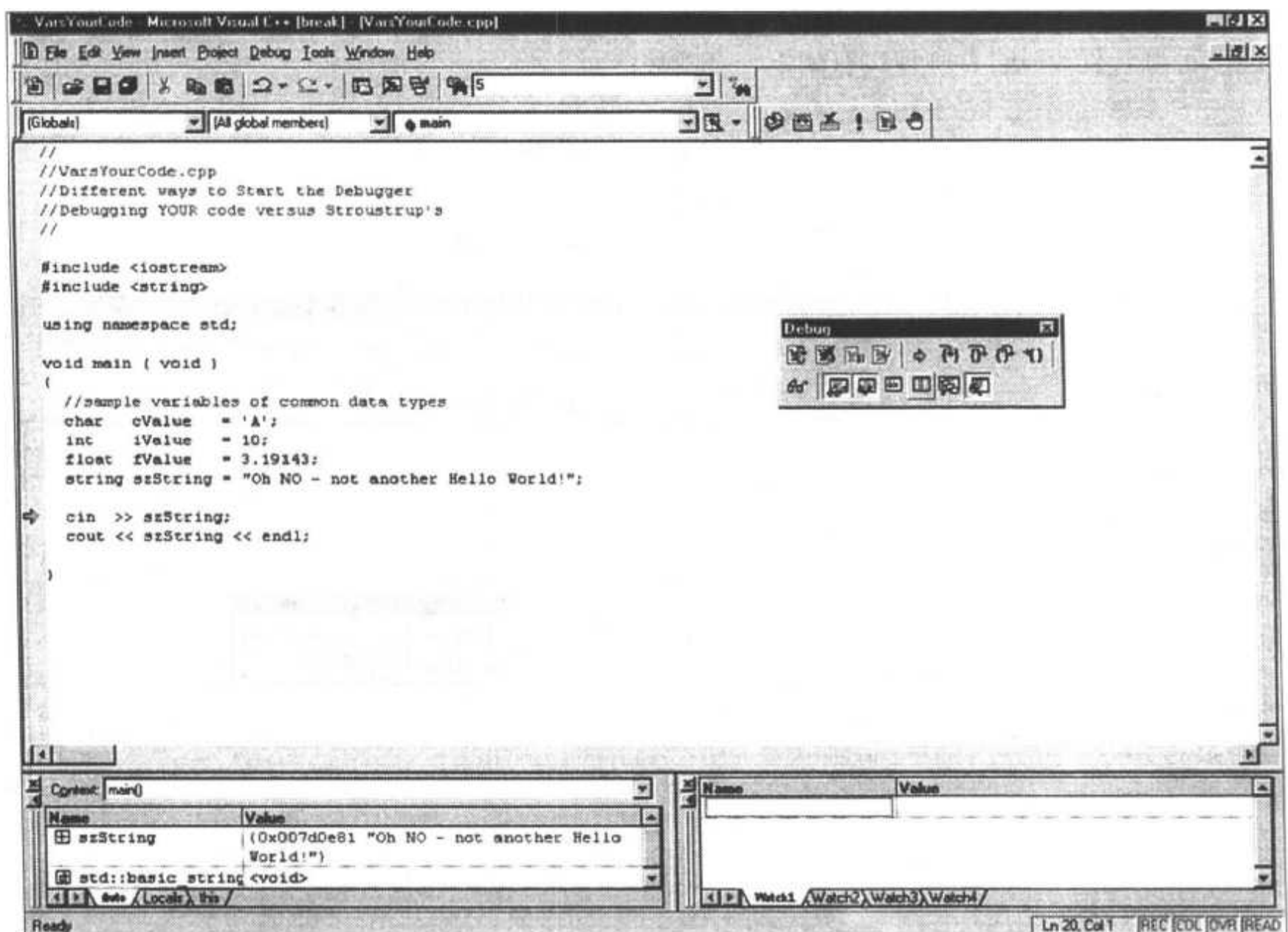


图 6-10 再运行 3 个单步跟踪后 Visual C++窗口的内容

使用 Auto 标签观察时存在一个问题，即其自动跟踪的范围过狭窄。此时，我们可以单击 **Variables** 窗口的 **Locals** 标签，观察所有的局部变量。

然而，即使这样也还存在一个问题。缺省的 **Variables** 窗口尺寸过小，无法观察到所有的局部变量。对于这一问题，一种解决方法是缩小代码页窗口，扩大 **Variables** 窗口，如图 6-11 所示。

6.1.2.1 扩大 Variables 和 Watch 窗口

为扩大 **Variables** 和 **Watch** 窗口，只要将光标放置于代码页框架的底边上，按住鼠标的左按钮，并向屏幕的上方移动鼠标，直至方便的位置即可(参见图 6-11)。

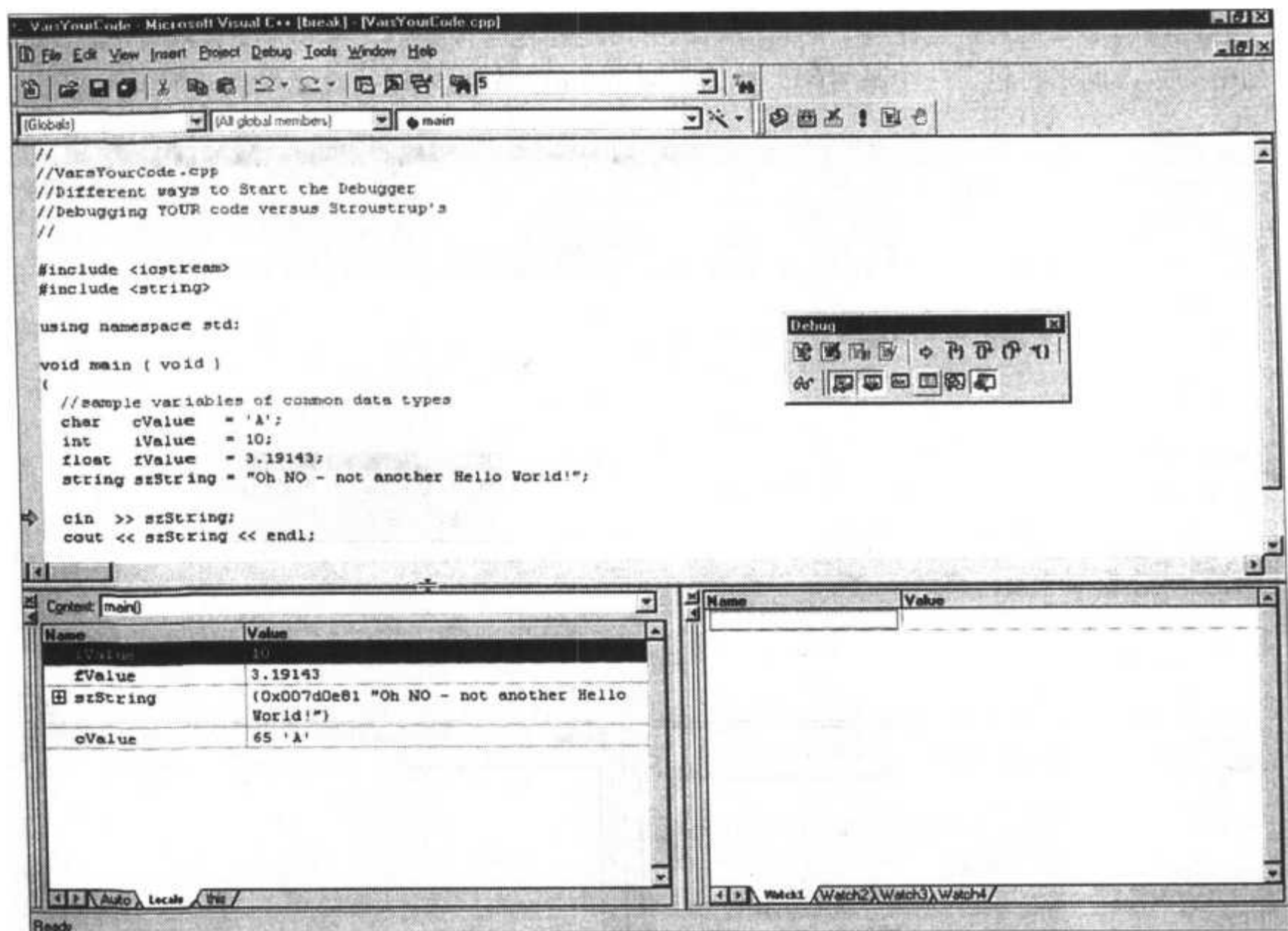


图 6-11 扩大的 Variables 和 Watch 窗口

6.1.2.2 理解 Variables 窗口内容

Variables 窗口中包含我们可能需要知道的关于变量的所有信息。每一个标签中都包含一个电子表格，该电子表格中有三个可调整尺寸的列。Debugger 将使用标签中变量的类型、名字及值，自动填充这些列。图 6-12 演示了 Variables 窗口对于 `szString` 字符串类型变量的一些功能。

位于标签上部的工具栏含有一个下拉列表，用于指定当前显示变量的范围。使用快捷菜单，可以将这一工具栏隐藏或重新显示。

新的 Visual C++ Variables 窗口使用新功能替换了以前 Visual C++ 版本的 Locals 标签。如果 Variables 窗口包含数组、对象或结构变量，则在变量名旁边出现一个按钮。通过单击该按钮，可以展开或折叠变量的观察视区。当变量以折叠形式显示时，按钮显示为加号(+); 当变量以展开形式显示时，按钮显示为减号(-)。通过单击这一加号框，可以展开对应的变量，



使其以树的形式展开，这种展开的树型结构中可能还包含按钮框。当一个变量展开后，Name 列中的按钮框中将包含一个减号(-)。单击带减号(-)的按钮框可以折叠已经打开的变量。

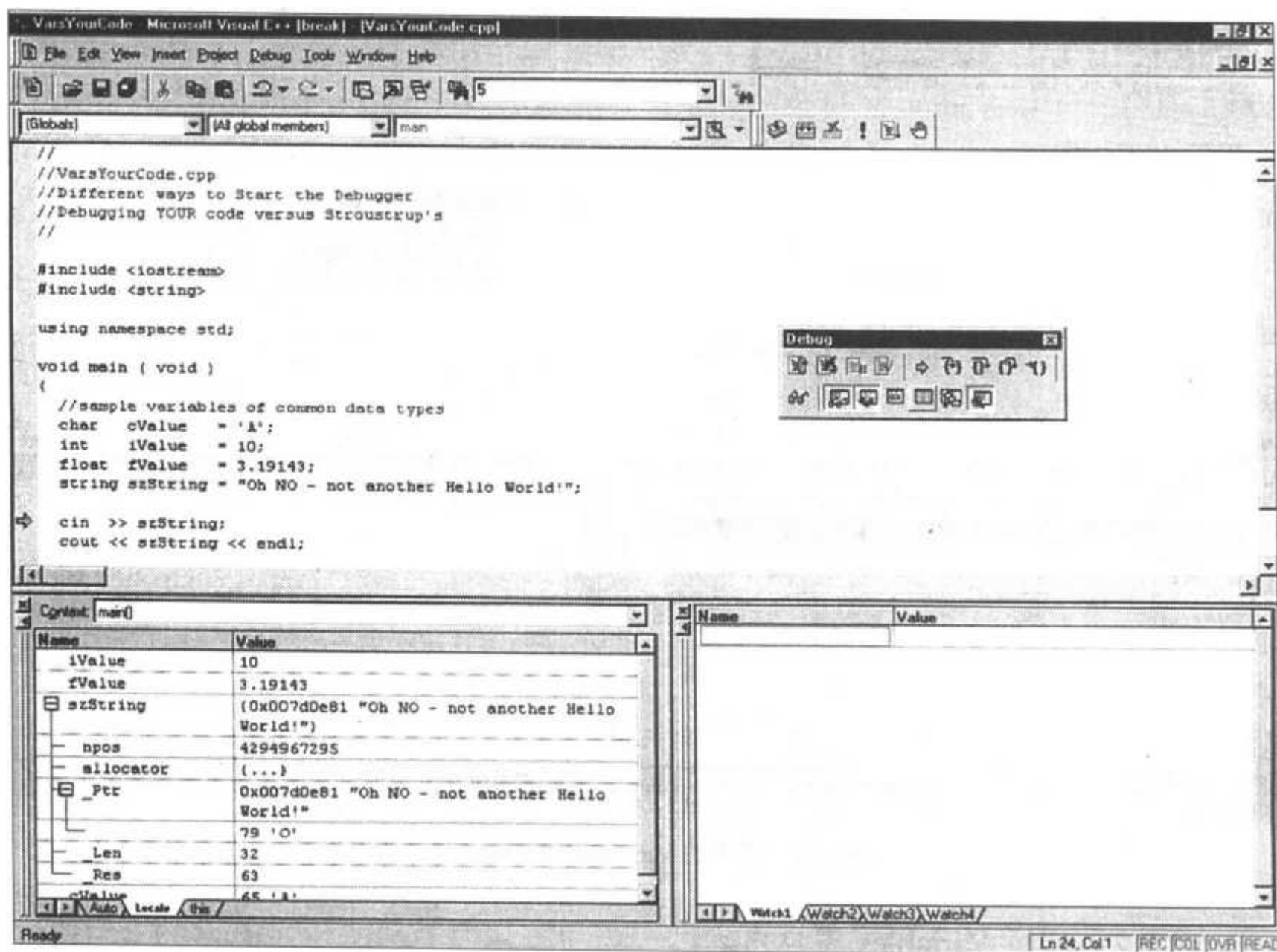


图 6-12 在 Variables 窗口的 Locals 标签中观察 szString

作为一种选择，通过选择一个变量后按加号或右箭头键，可以展开该变量。通过选择一个变量后按减号或左箭头键，可以折叠该变量(下一节中将要讨论的许多选项，将在本书以后各章中的适当时候展开介绍)。

Variables 窗口支持编辑。用户可以剪切、复制或拖动 Variables 窗口中的信息。可以在调试时编辑 Value 列改变一个变量的值。如果在 Options 对话框(Tools 菜单中)的 Debug 标签中选择了十六进制显示方式，则必须输入十六进制的值或使用前缀“0n”(零-n)指明该数为十进制数(例如，0n1000)。

也可以重新配置行列的行为。要自动调整列宽使其适应于其中的内容，双击列边界处



的垂直分界线即可。为手工调整列宽度，向左或向右拖动其右边的分界线即可(行的高度正好适合于当前字体的大小，不能手工调整其高度。为改变字符大小，使用 Options 对话框[Tools 菜单中]的 Font and Colors 标签即可)。

Auto 标签显示关于当前语句和前面的语句中变量的信息。变量的出现顺序以字母顺序排序。如果一条语句占用了多行，则 Auto 标签显示的变量从该语句对应的行开始，最多可达 10 行。

Locals 标签显示当前函数中所有本地变量的名字、值及类型。在跟踪遍历一个程序时，新的变量将进入活动范围。

This 标签显示 This 指针所指向对象的类型、名字及值信息。所有对象的基类将自动展开。

设计提示

右击 This 窗口可以弹出一个命令快捷菜单。

6.1.3 小心调试代码

当跟踪箭头处于一个代码语句，该代码语句激活一个方法(有时称为成员函数)或一个标准 C/C++ 子程序的独立函数时，必须十分小心。此时，必须明确是选择 Step Into 还是 Step Over 选项。

图 6-12 中，跟踪箭头处于 `cin>>szString;` 语句，此时，我们需要使用 Step Over 调用 `cin`，无须调试 `cin` 本身。

如果读者正在执行 `VarsYourCode.cpp`，则可以按 **F10**，或单击 **Debug 工具栏** 上的 **Step Over 按钮**(第一行从左起第 7 个按钮，参见图 6-4)。此时，将看不到任何明显的结果发生(实际上，如果仔细观察，可以看出 Variables 窗口中的下拉范围列表变成了灰色)。

6.1.3.1 如何在调试一个应用程序时输入用户输入信息

当调试一个期待用户输入信息的代码语句时，需要切换到执行窗口。Windows 的任务切换组合键 **ALT+TAB** 是在 Debugger 窗口和 MS-DOS 模式兼容窗口之间切换的最快方法。此时，**按 ALT+TAB**，则切换到执行窗口(参见图 6-13)。**再按 ALT+TAB**，可返回到 Visual C++ 窗口。当然，这种方法只有在没有其他正在运行的任务时才有效，或者没有首先在 Microsoft Visual Studio C++ 与其他任务之间执行任务切换。

为了输入新的字符串，第三次**按 ALT+TAB**，切换回执行窗口，**输入 “NewString”**(不包括引号)后**按 ENTER 键**，如图 6-14 所示。注意跟踪箭头(参见图 6-13)仍然位于 `cin>>` 语句处。

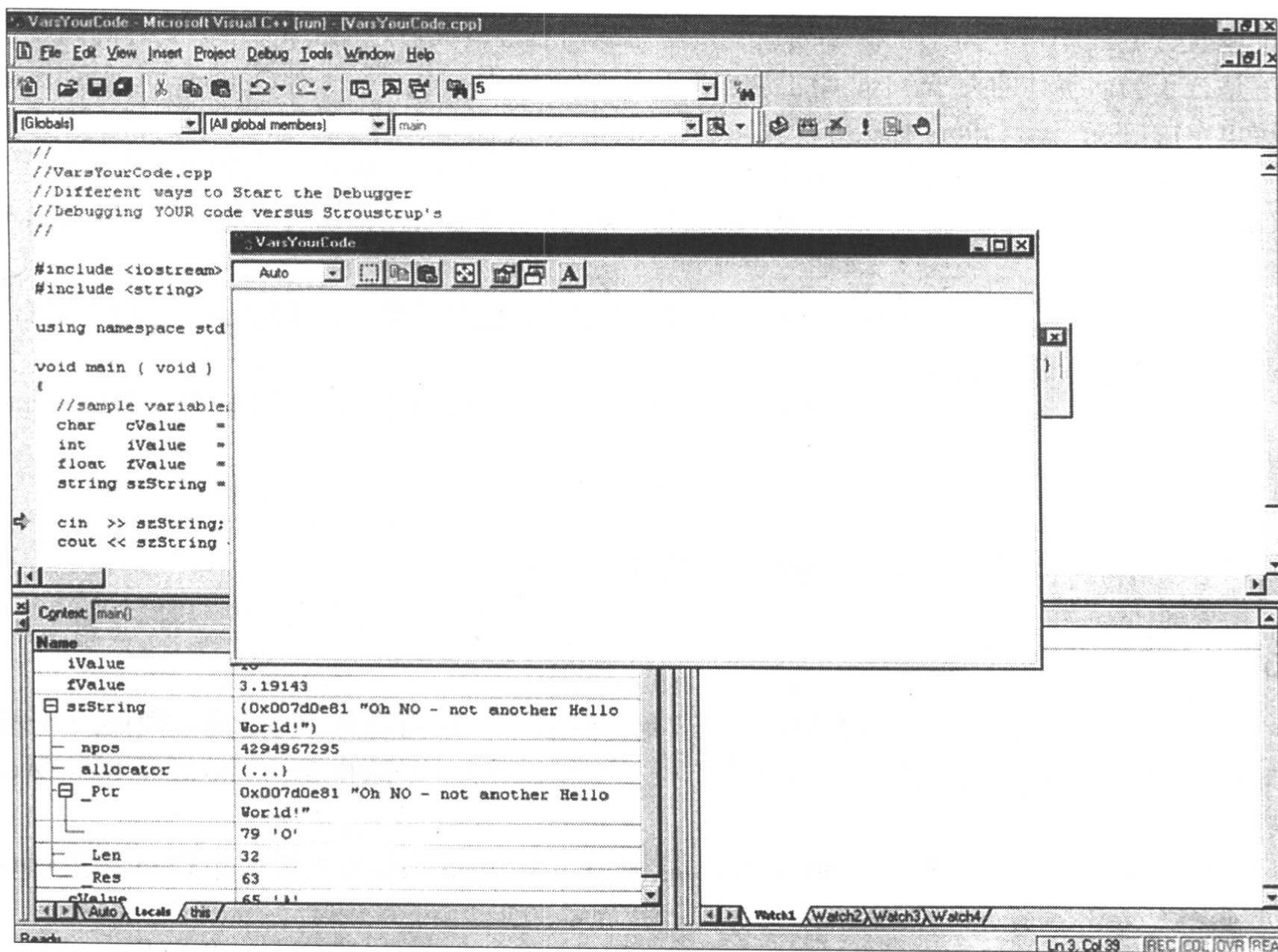


图 6-13 按 ALT+TAB 任务切换到 MO-DOS 执行窗口

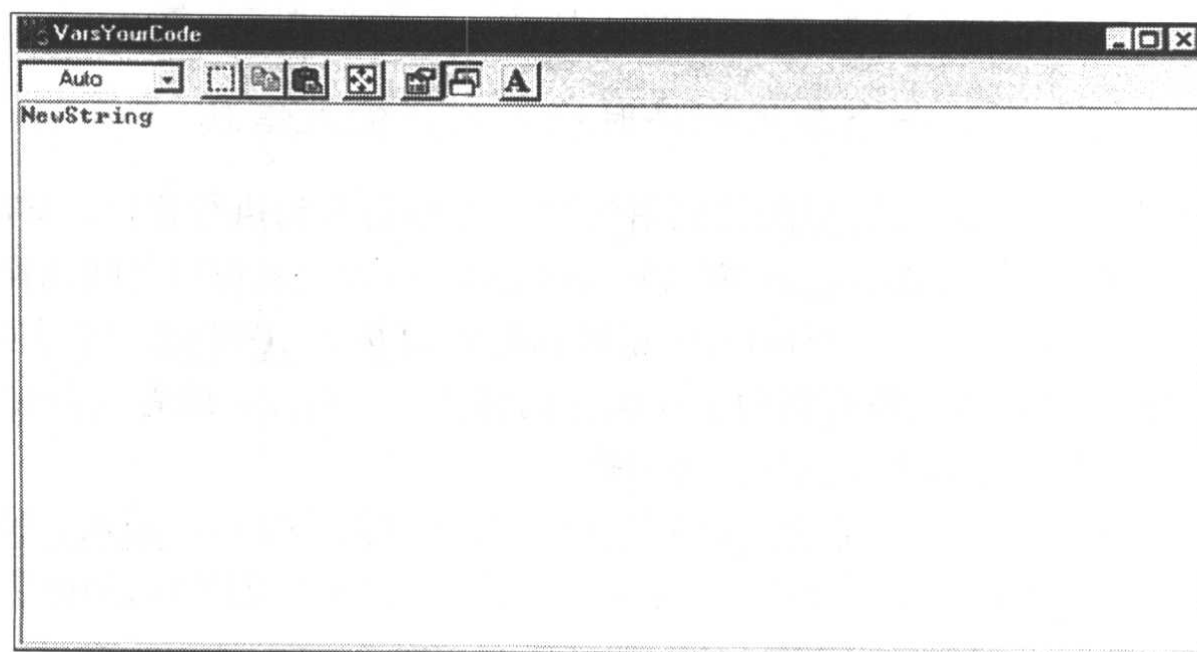


图 6-14 调试一个应用程序时与用户交互

一旦输入 *NewString* 后按了 ENTER 键，则显示屏幕应类似于图 6-15 所示，Debugger 自动切换回主 Visual C++ 窗口。

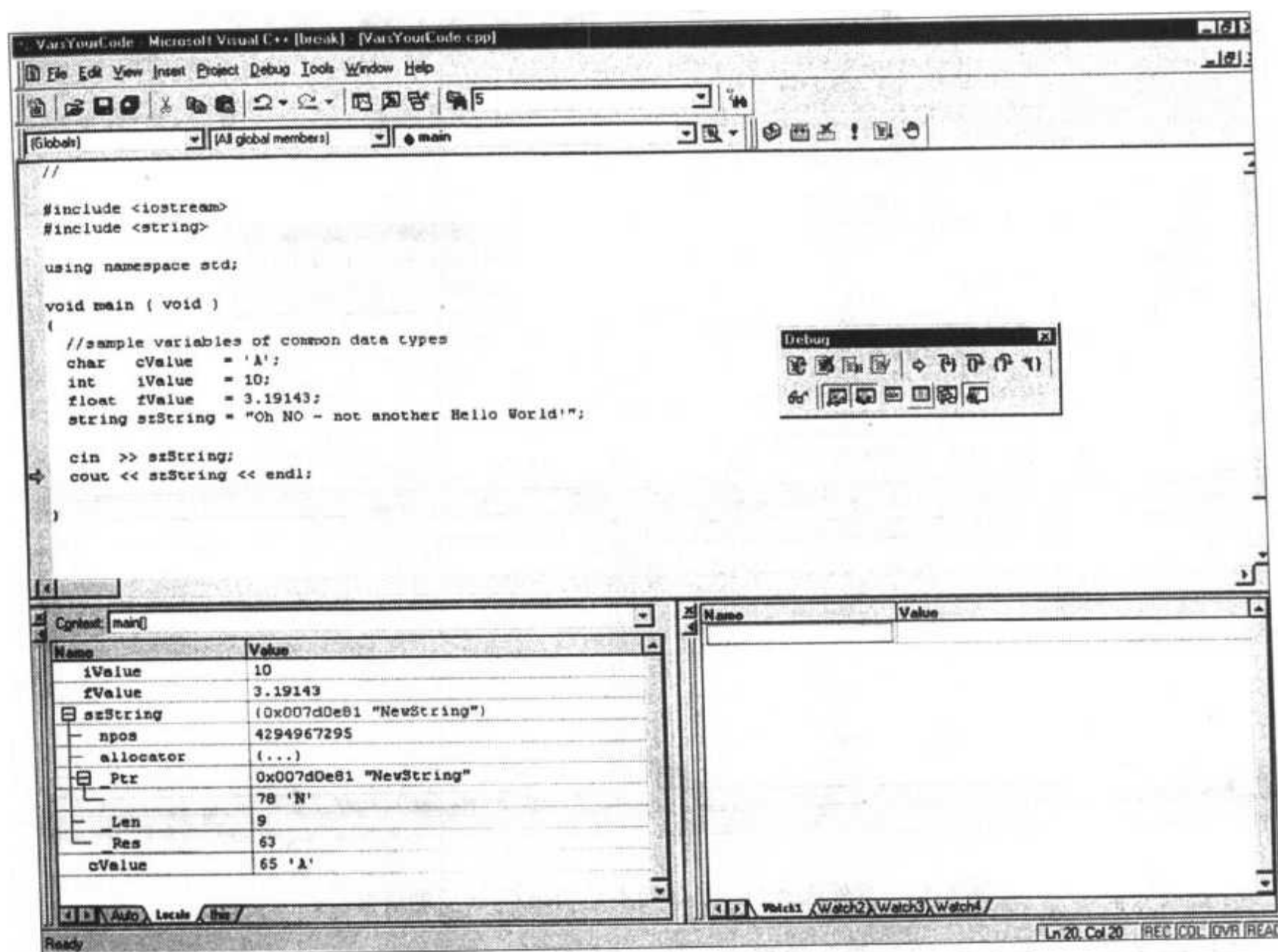


图 6-15 新输入的 szString 值更新了 Variables 窗口

虽然 Debugger 使用不同的颜色表示了 Variables 窗口中的内容，但在黑白图像界面下这些区别是很难发现的。然而，包含了新输入的“*NewString*”的 *szString* 的值以红色显示。任何时候当一个变量的值由前面执行的代码语句改变时，Debugger 将改变 Variables 窗口中 Value 字段的颜色。

如果在跟踪箭头位于 `cin>>szString;` 语句时已经选择了 Step Into 选项，则可能已经看到了如图 6-16 所示的一个 Debugger 窗口。

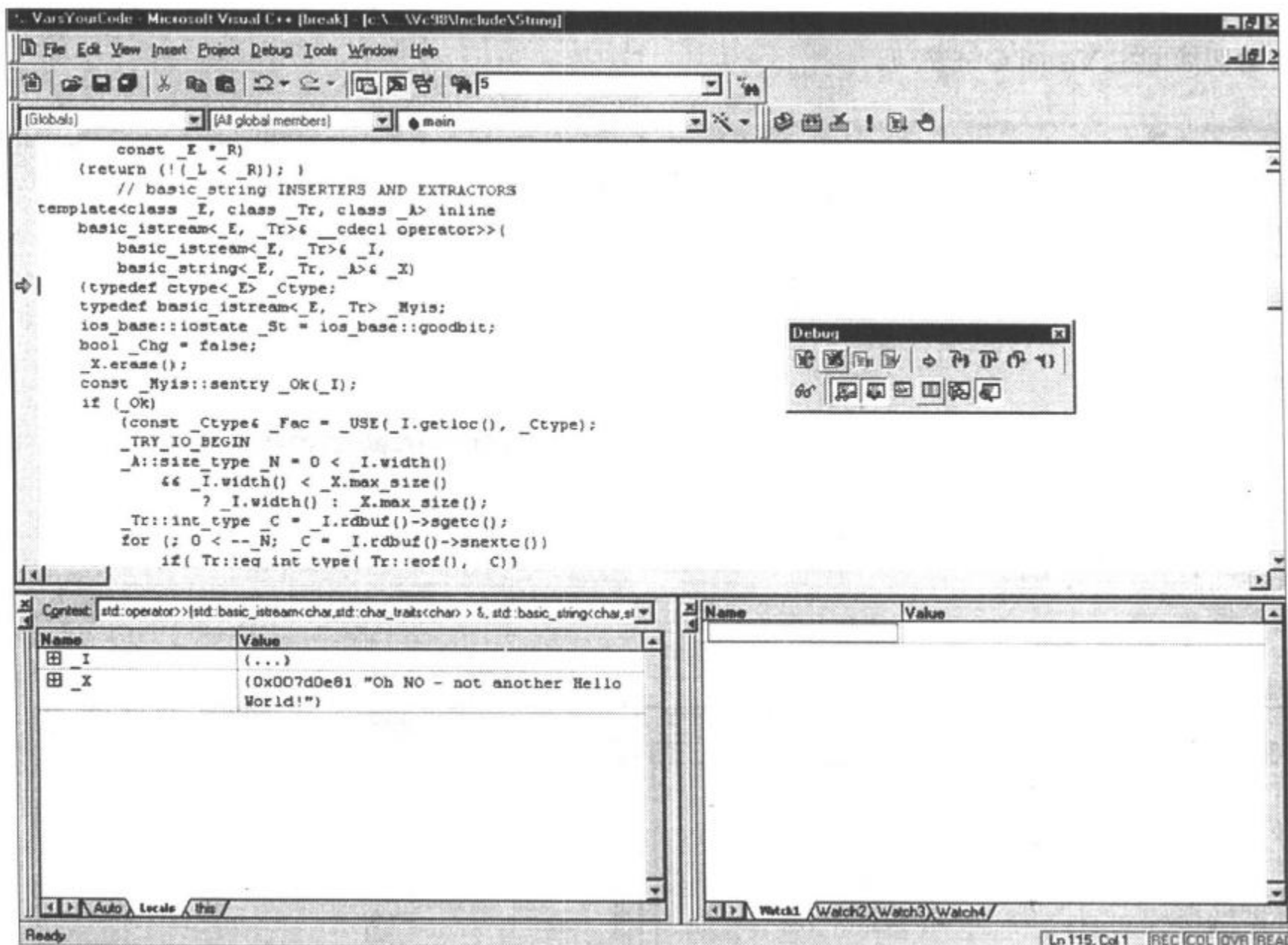


图 6-16 意外选择 Step Into 而不是 Step Over 后的情况

Visual C++ Debugger 的功能非常强大，这一例子演示了 Debugger 调试语言相关程序的能力。此处，我们看到 Debugger 的窗口内容已经变成了 `szString` 的 `string`，`basic_istream` 模板，并准备逐行调试该模板。

当无意中选择了 Step Into 而不是原本打算选择的 Step Over 时，应选择 Step Out 选项立即返回到自己的程序代码中，Step Out 选项是 Debug 工具栏(参见图 6-4)中第一行左起第 8 个按钮，或按 SHIFT+F11。

6.1.4 快速查看变量的内容

Debugger 的一个巧妙特性是，只要将 I 鼠标指针悬浮于源代码中一个变量的名字上，即可显示该变量的当前内容。图 6-17 给出了 I 鼠标指针处于 `cout<<szString;` 语句中 `szString` 上的情况。

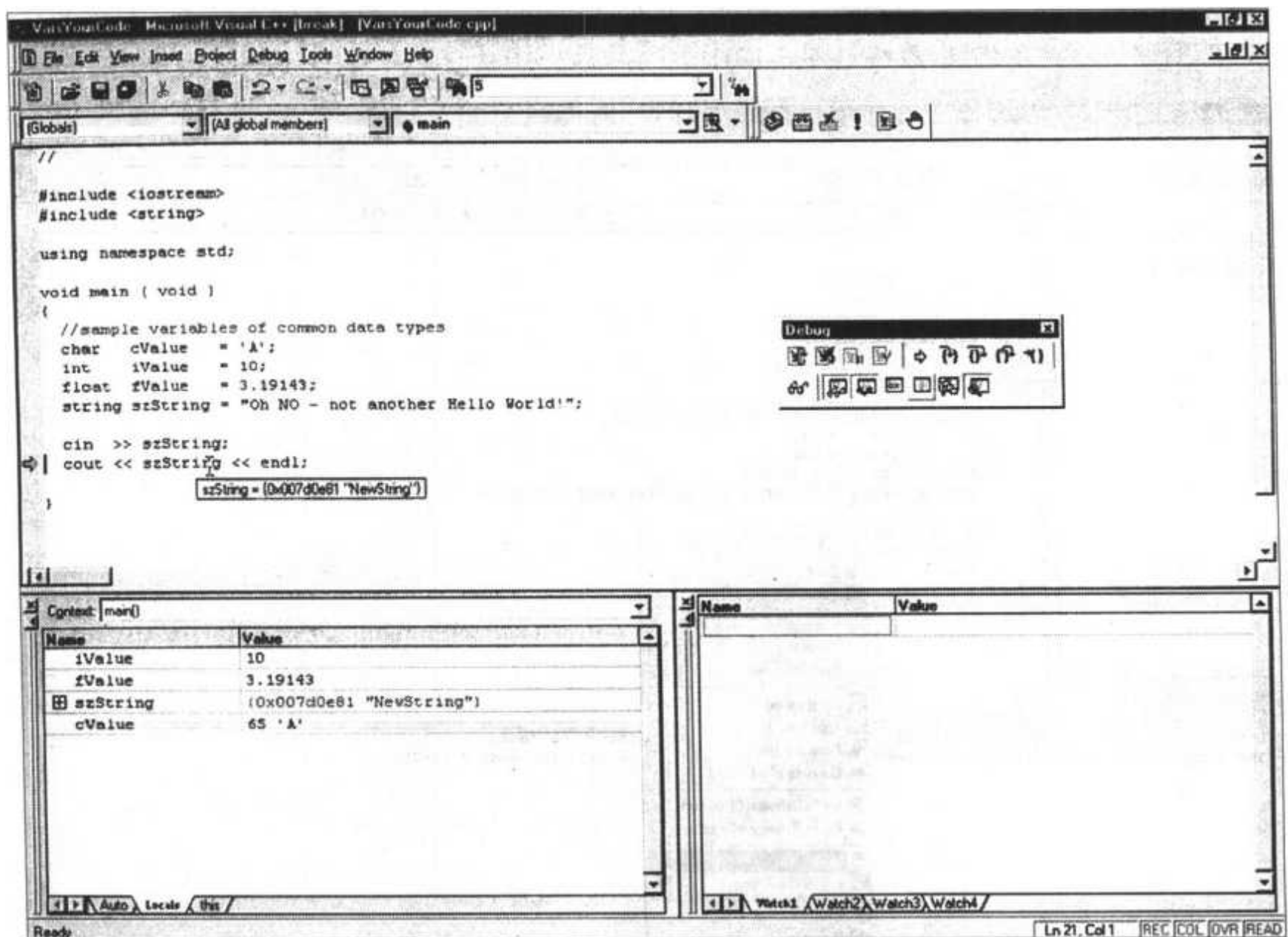


图 6-17 使用 I 鼠标光标查看变量当前内容

弹出窗口中显示了变量的名字、当前值以及数据类型——更详细的是此例中 *szString* 的起始地址(十六进制)。

6.1.5 中途停止 Debugger

由于这样或那样的原因，有时候需要停止 Debugger 调试。当然，一种原因是最后单步执行的结果反映出程序中存在一个逻辑错误。为了停止 Debugger，只要按 SHIFT+F5，或单击 Debug 工具栏中第一行左起第二个图标即可。如果读者正在 Debugger 调试本实例程序，则此时需要停止 Debugger。

6.1.6 执行到代码的指定行

当调试一个包含以前已经调试过的代码段的程序时，不需要再单步调试那些“好的”



代码。在这种情况下，需要设置一个断点。此时 Debugger 可以全速执行该程序，直至断点处，然后停下等待单步调试(参见图 6-18)。

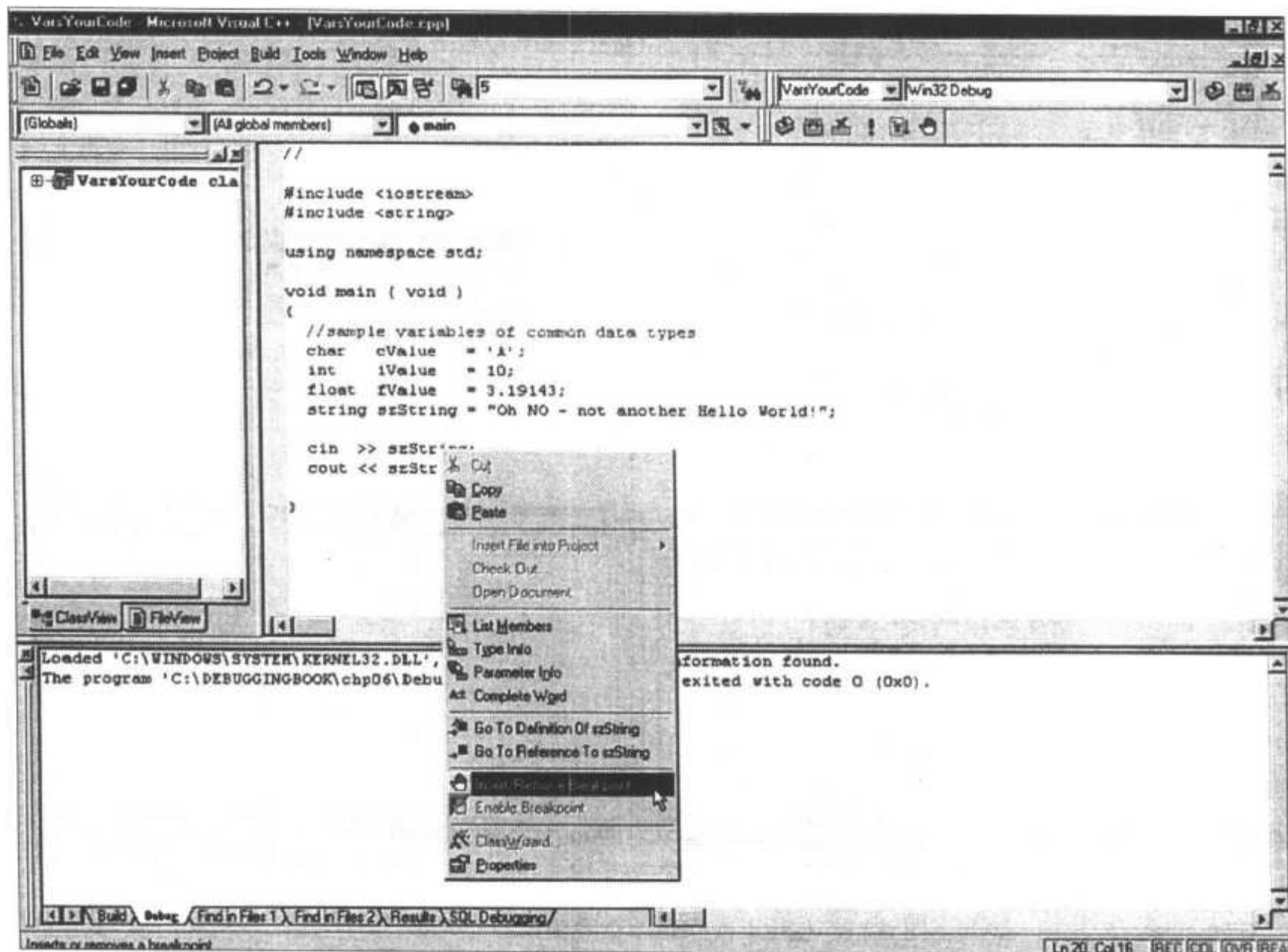


图 6-18 通过将光标置于一语句后右击设置断点

设置断点最快的方法是：

1. 将光标放置到需要 Debugger 停止的代码语句上。
2. 单击 Insert/Remove Breakpoint 选项(执行这两步可以删除以前已经设置的断点)。

现在在交互式跟踪中，使用所喜好的方法，在 cin>>语句设置一个断点。正如在图 6-19 中所看到的那样，通过查看任何语句的左侧是否有(红色)停止符号图标，可以很容易地知道一个程序是否设置了断点。

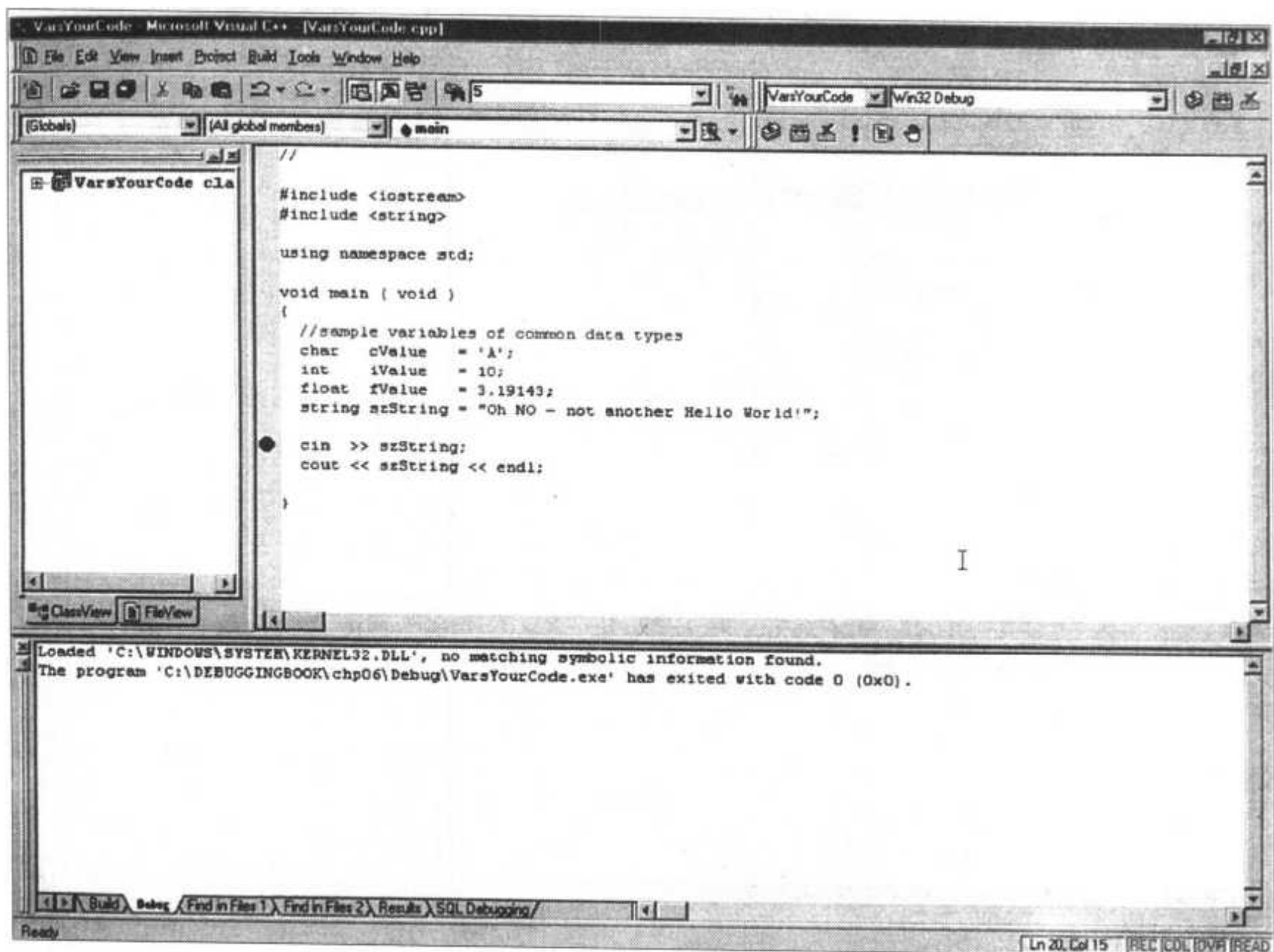


图 6-19 查看(红色)停止符断点设置

6.1.7 全速执行到一个断点

一般来说，有多个界面选项可以用来全速执行程序直至一个断点。最长的步骤是，首先单击 Visual C++ Build 主菜单选项。然后选择 Start Debug 选项，再选择 Go 选项(参见图 6-20)。最快的方法是按 F5 键。

如果读者正在按照本例使用 Debugger，现在按 F5。正如在图 6-21 中所看到的那样，跟踪箭头现在直接放置在了断点(红色)停止符上。Variables 窗口中包含了所有已经定义和初始化的变量的内容，正如单步执行了每一条声明语句一样。

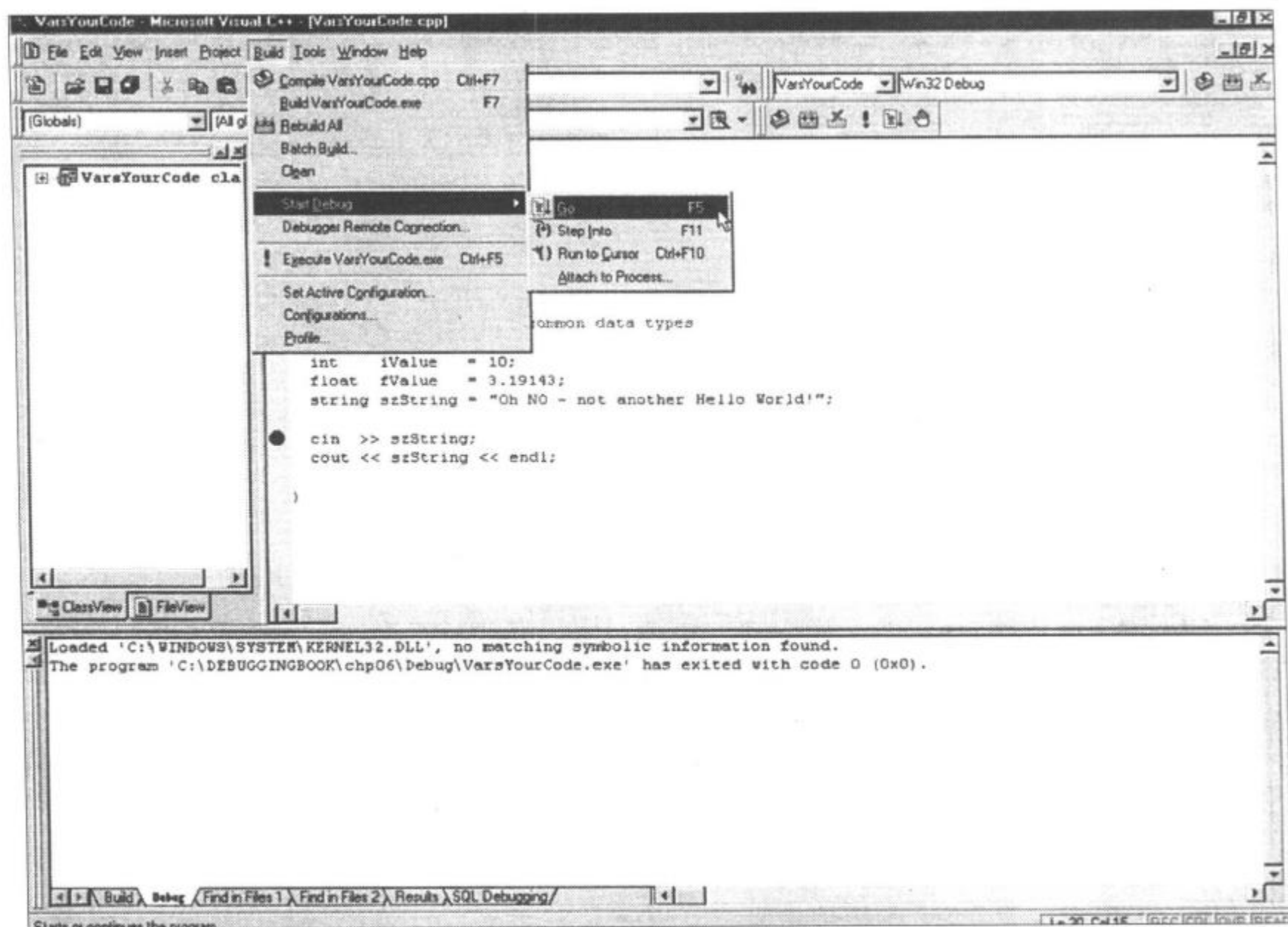


图 6-20 访问 Go(F5)命令的最长操作方法

6.1.8 运行至光标处

虽然在此处没有演示，但全速运行到一个指定代码语句更快的方法是使用 Run to Cursor 选项。Run to Cursor 执行程序直到包含插入点的代码行(换句话说，即源代码中放置 I 光标代码处)。这与在插入点位置设置一个临时断点等价。

当 Debugger 运行时，可以通过单击 Debug 菜单再选择 Run to Cursor 选项，或同时按 CTRL+F10 键，访问 Run to Cursor 命令。

6.1.9 现在测试

由于本书的目的是调试应用程序，所以本书假设读者正处于一个调试过程中，并且现在已经做好了全速执行全部“调试过”算法的准备。Go 命令实际上有三种用途：

- 全速 Go(执行)到一个断点，并等待单步调试。



图 6-21 带有断点运行 Go(F5)后的结果

- 全速 Go(执行)到下一个断点，并等待单步调试(如果设置了多个断点)。
- 全速执行程序。

如果读者一直在按照例子操作，则按 **F5**(或使用图 6-20 中讨论的多步操作法——不存在任何偏见)完成 VarsYourCode.cpp 的执行。

6.2 高级 Debugger 技巧

在应用程序的开发周期中，即使知道了上面所讨论的基本 Debugger 命令，仍不能保证在调试阶段执行最有效地调试。除了知道 Debugger 可以做些什么和如何机械地激活这些选项之外，使用 Debugger 还有更多的技巧。

本章的后半部分将开始讨论若干个更高级的 Debugger 选项，并在最后讨论如何更灵活地使用 Debugger 的功能，以在最短的时间内完成调试。



6.2.1 使用新值运行

Microsoft Visual C++ Debugger 包含许多节省时间的特性。例如，Debugger 从当前语句开始，如同变量具有不同的值一样，继续调试一个应用程序的功能。一个快速例子是测试 `if...else` 语句：

```
if (cResponse == 'Y')
    cout << "True action taken...\n";
else
    cout << "False action taken...\n";
```

Debugger 提供了三种在调试时修改变量内容的方法。

6.2.1.1 在 Variables 窗口中修改一个变量的值

由于使用自动跟踪检查变量的内容是一种容易且可行的手段，所以输入变量的新值最容易的方法就是在 Variables 窗口内。如下的五步解释了具体操作方法：

1. 首先单击 Variables 窗口内的 Auto 标签、Locals 标签或 This 标签。
2. 然后单击需要修改的变量。
3. 当该变量是一个数组或对象时，使用+框展开，直到看到需要修改的值。
4. 任务几乎已经完成了。现在双击该值，或使用 TAB 键移动插入点到需要修改的值。
5. 最后，输入新值后按 ENTER 键，新值将显示为红色。

现在可以继续调试应用程序，如同该变量原来就包含该新值一样。

现在，如果读者要一起交互式跟踪样这个程序，则需要输入如下的程序并单步跟踪直到跟踪箭头处于 `if...语句`。

```
//
//RunTimeValueChanges.cpp
//Different ways to reset variable contents
// at run time
//
#include <iostream>
using namespace std;
void main ( void )
{
    char  cResponse = 'Y';
    if ( cResponse == 'Y' )
        cout << "True action taken...\n";
    else
        cout << "False action taken...\n";
}
```

图 6-22 给出了一个初始 Debugger 会话，所有这些都是为调试 *cResponse* 变量的初始化为“Y”的一个 **if...else** 语句。注意，跟踪箭头位于 `if(cResponse=='Y')` 语句，并且 Variables 窗口以所希望的“Y”值显示 *cResponse*。

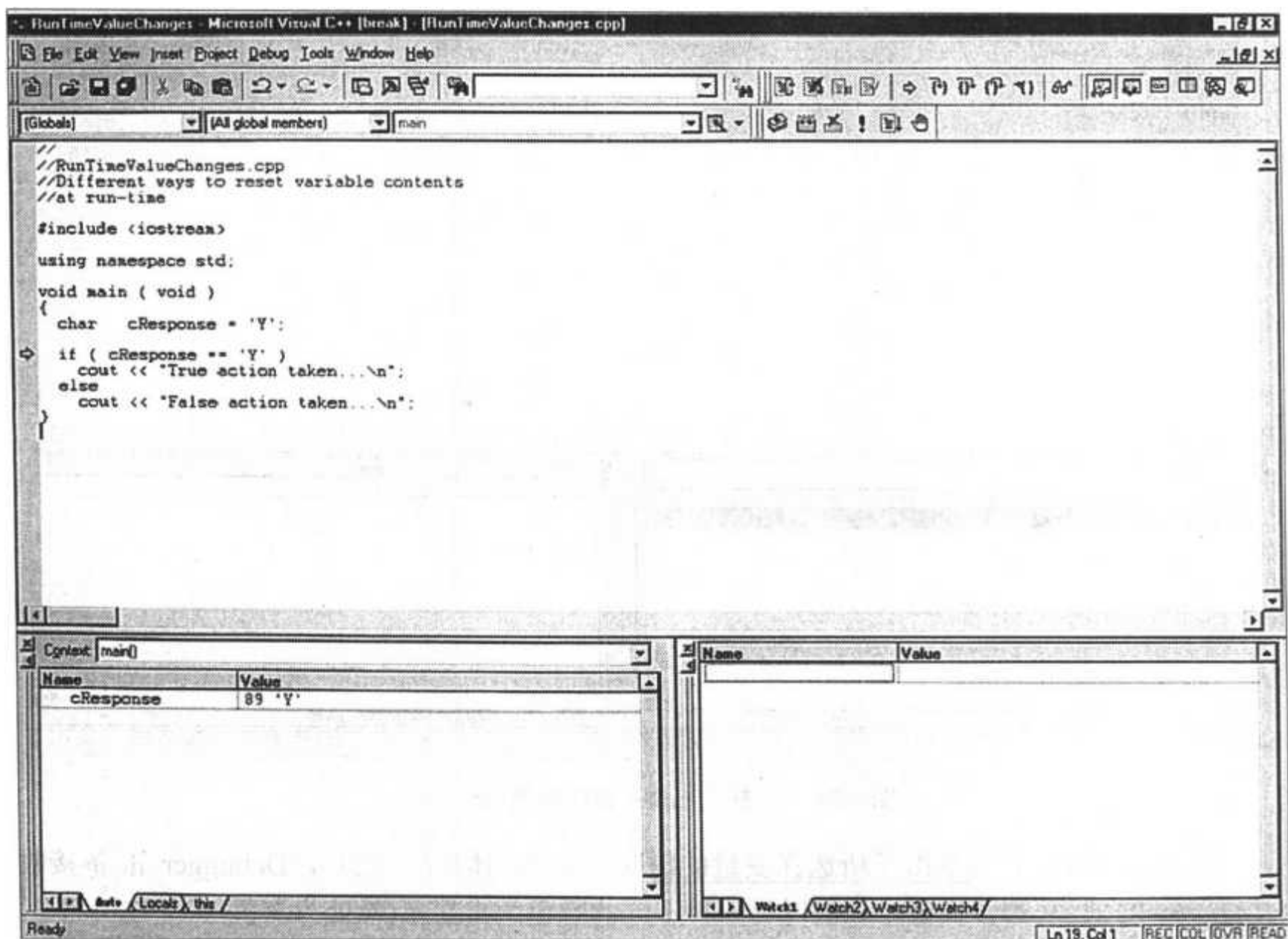


图 6-22 设置 Debugger 调试 *cResponse* 值‘Y’

(作为一种变化，图 6-22 中给出的 Debug 工具栏移动到了 Visual C++ 窗口的主菜单区。通过拖动 Debug 工具栏的标题到主菜单区域即可。Visual C++ 将自动调整 Debug 工具栏的窗口形状，使其适合于主菜单区域)。

图 6-23 给出了在 Variables 窗口内单击一个变量的名字时所发生的情况。注意，Debugger 已经将显示的“Y”转换为了其对应的 ASCII 表值 89(由于 Debugger 彩色语句条的原因，要看清楚有些困难)。

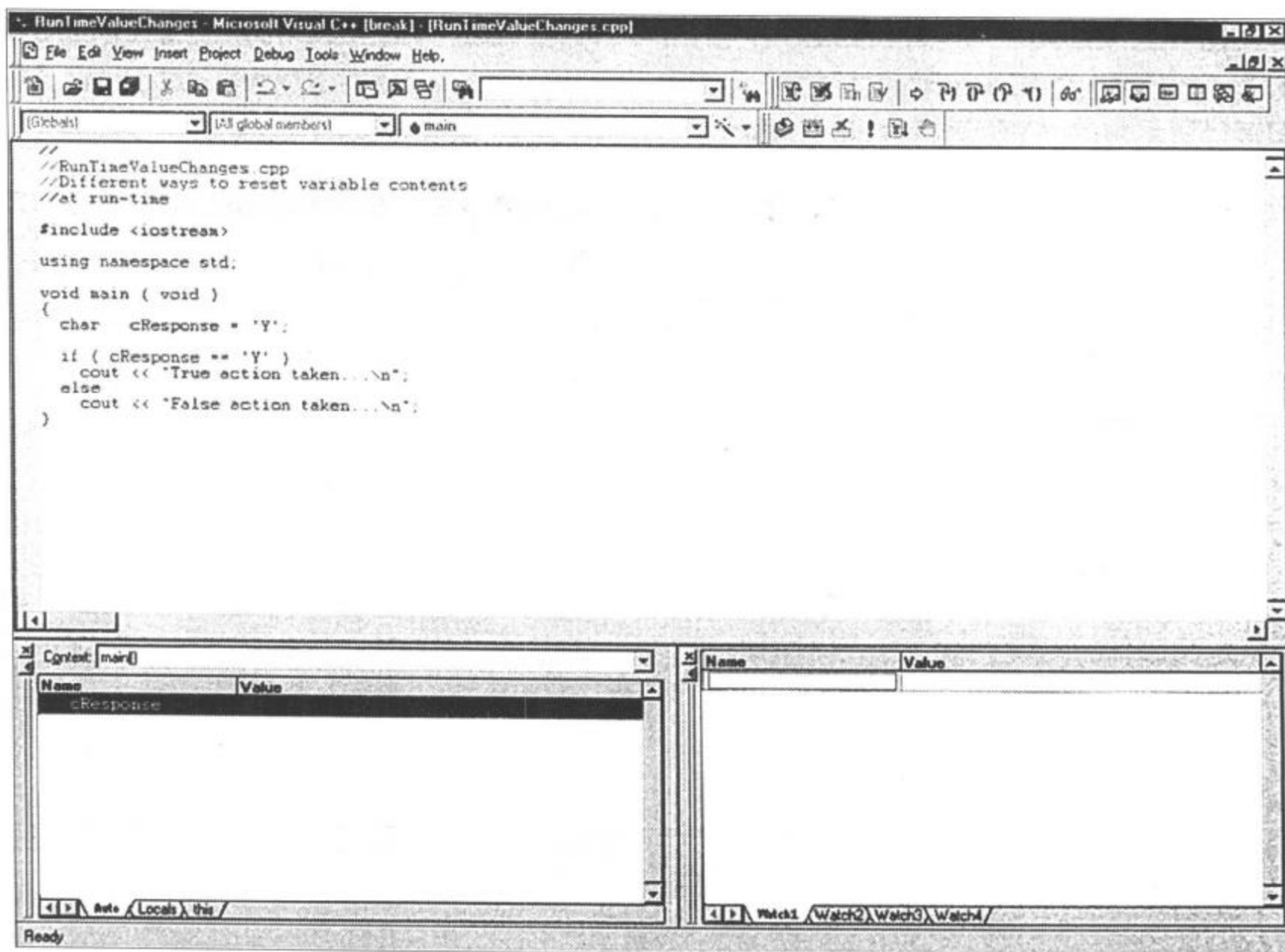


图 6-23 单击 Variables 窗口内的一个变量

图 6-24 演示了当单击了所选择变量的值时所发生的情况。此时，Debugger 准备接收 *cResponse* 的值 89 的替换值，该值以另一种不同的颜色突出显示(此处为灰色)。

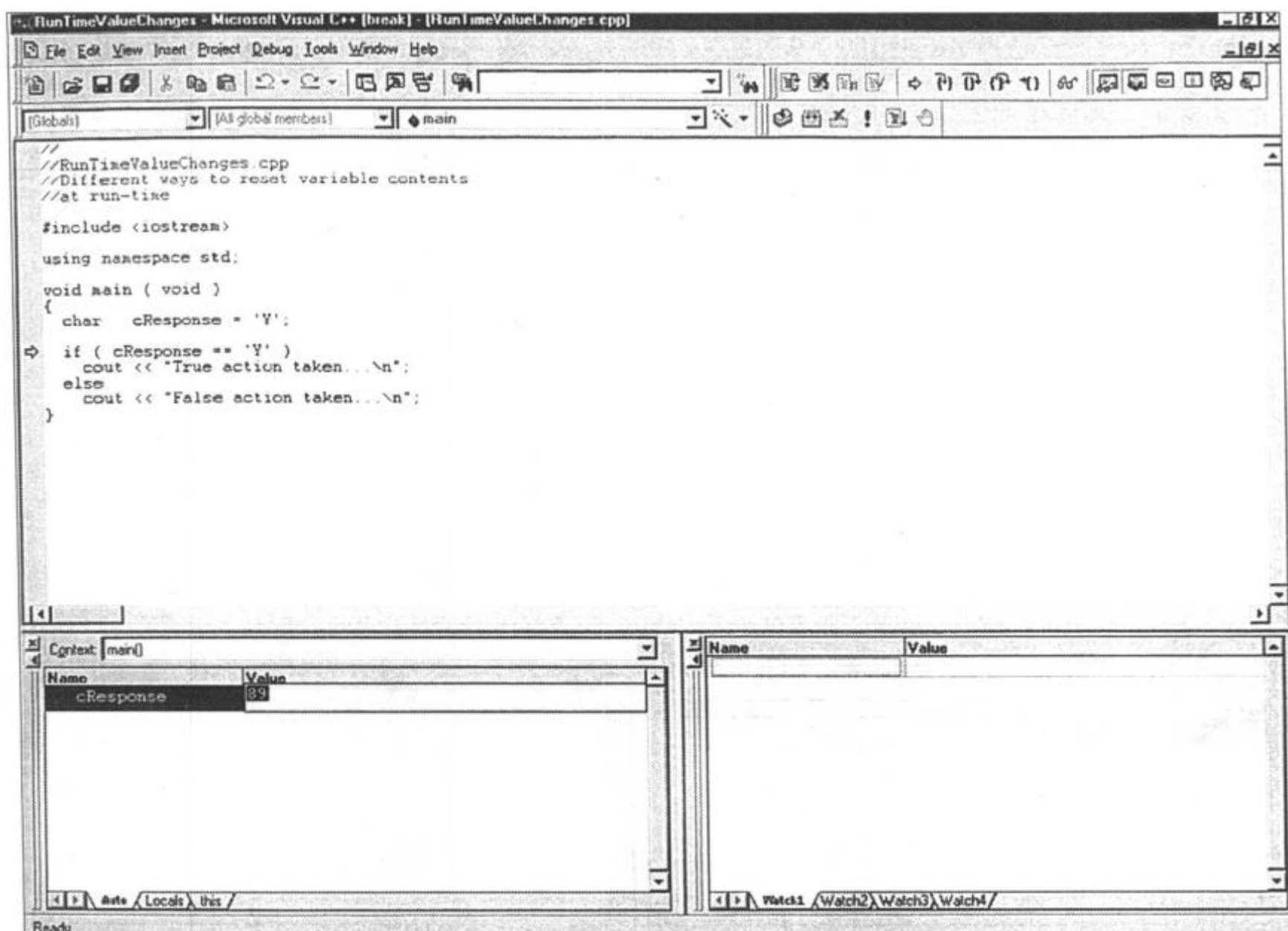


图 6-24 设置 Variables 窗口直到改变了变量的值

为了输入变量的新值，只要输入适当的数据类型和范围即可。图 6-25 给出了新的 `cResponse` 的 ASCII 表值 78，这与大写的“N”等价。

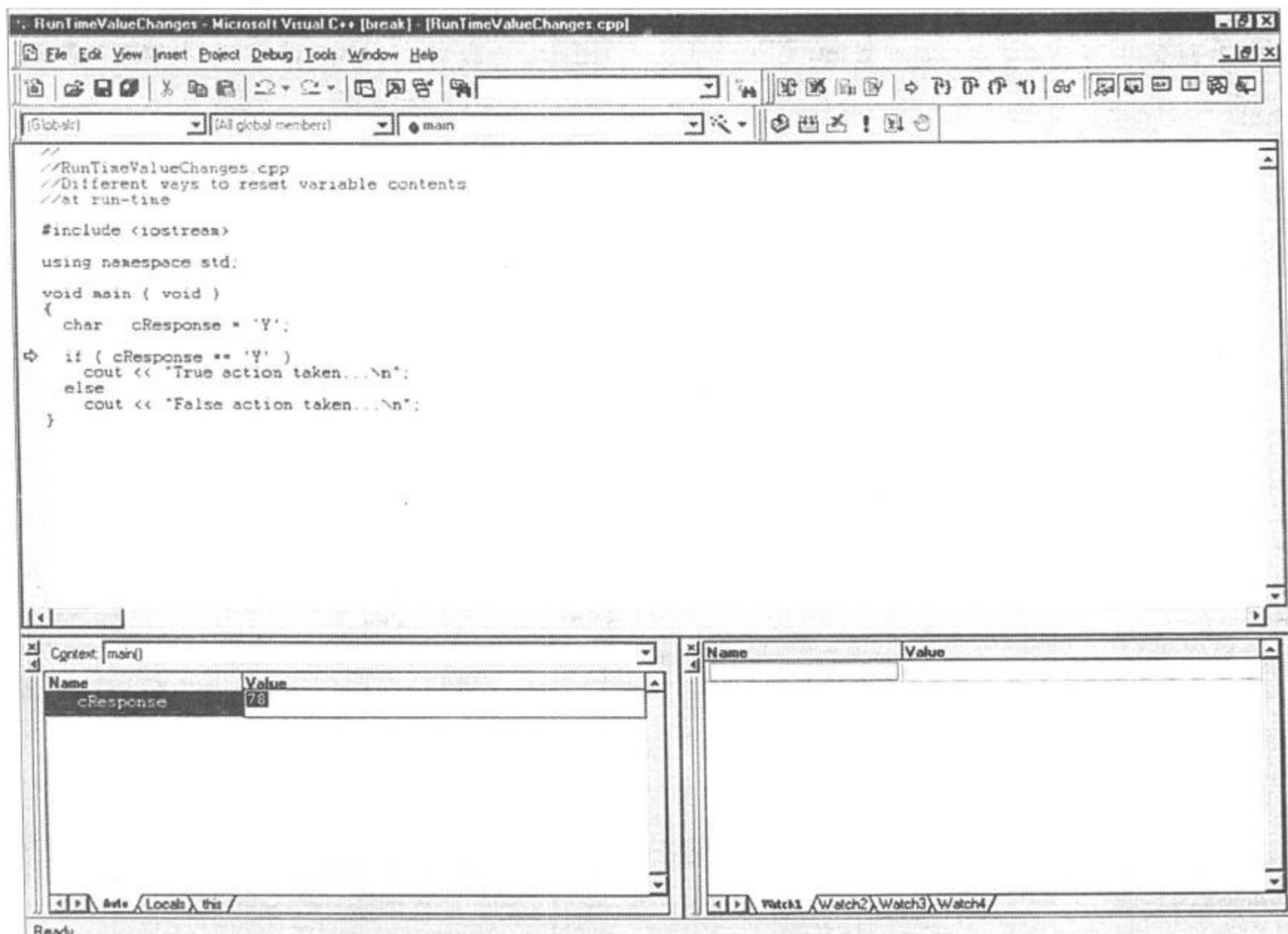
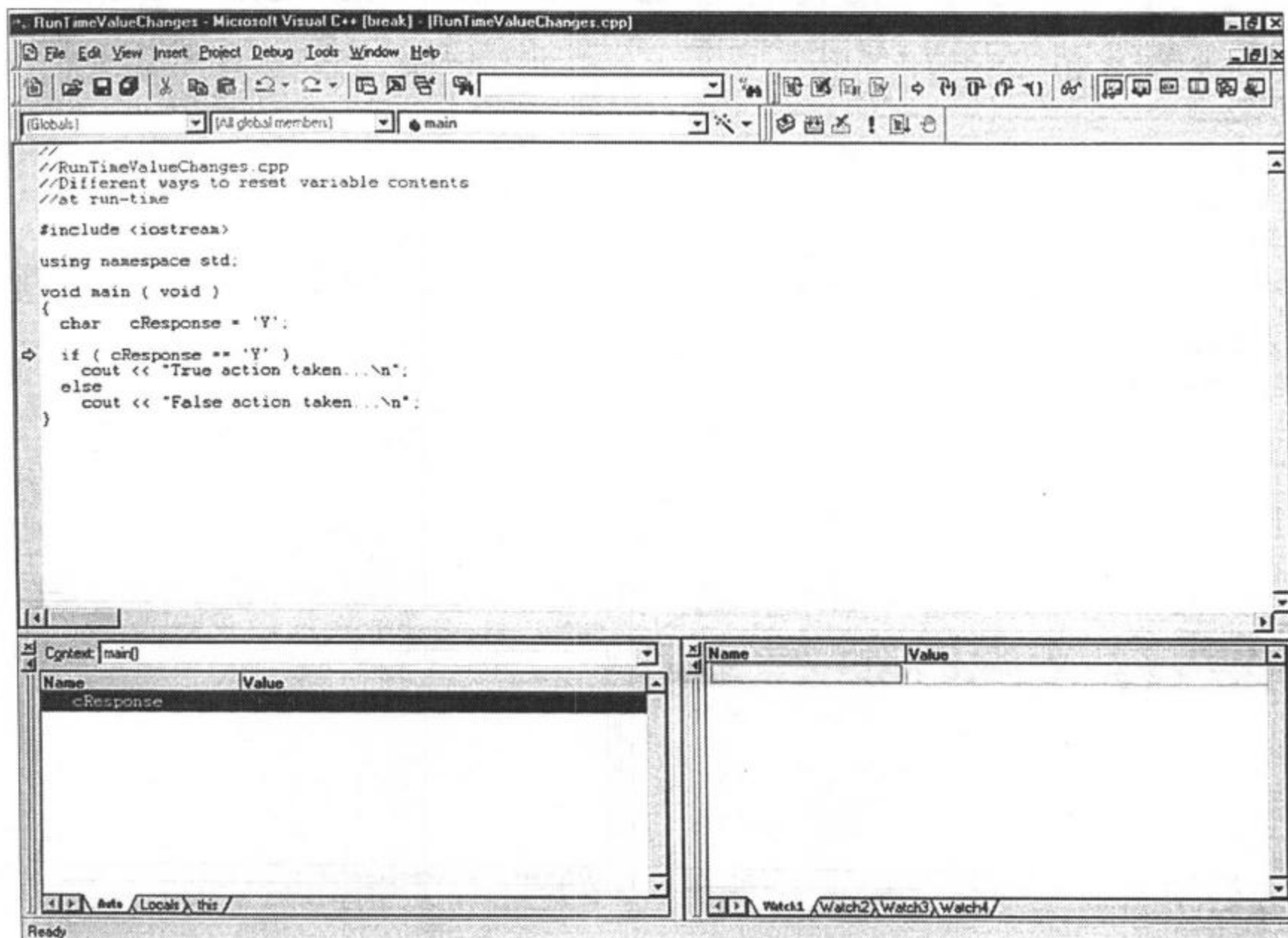


图 6-25 将 `cResponse` 的值从 ASCII 表的 89(“Y”)改变为 78(“N”)

图 6-26 通过调试 `if...else` 语句的 `else` 部分，证明了 Debugger 接收了 `cResponse` 的这一新值，调试的方法是按 **F10 键(Step Over)**。


 图 6-26 可以单步调试 `cResponse` 为 “N” 时的 `if...else` 语句了

从图 6-27 可以看出，跟踪箭头已经跳到了 `if...else` 语句的 `else` 部分的 `cout` 语句(要查看新输入的 `cResponse` 的值，需要单击 `Variables` 窗口的 `Locals` 标签，原因是 `cResponse` 的自动跟踪已经走出了所跟踪语句的范围)。

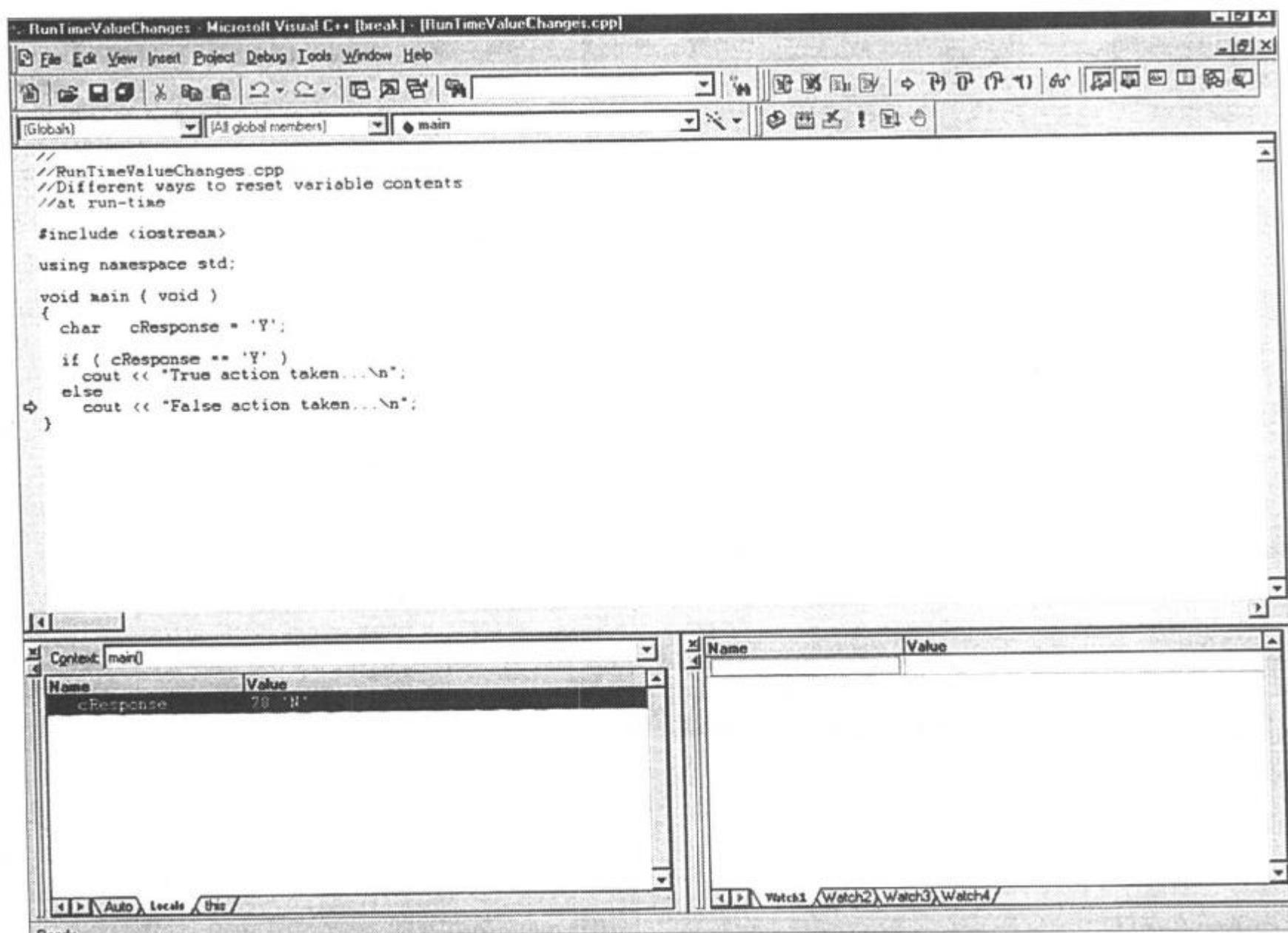


图 6-27 跟踪箭头说明运行时接收了“N”

对于用户来说，小心使用这种调试时修改变量值的方法，可以节省一个应用程序的大量调试时间，这样将无须强迫用户使用另一个不同的值重新初始化一个变量，更无须与一个新创建的外部文件(磁盘、网络或交互式终端用户)进行交互。

也可以从其他两处执行同样的变量值替换操作，即 Watch 窗口和 QuickWatch 窗口。

6.2.1.2 使用 Watch 窗口修改变量的值或寄存器的内容

为了在调试阶段使用 Watch 窗口修改一个变量的值，执行如下步骤：

1. 首先，双击 Watch 窗口中的变量，或使用 TAB 键移动插入点到所要修改的值。
2. 然后，如果变量是一个数组或对象，使用+框展开，直到找到所要修改的值。
3. 最后，插入新值后按 ENTER 键。

6.2.1.3 使用 QuickWatch 修改变量的值或寄存器的内容

调试阶段修改一个变量的值的第三种方法涉及到与 QuickWatch 的交互，下面的七个步骤说明了具体的操作方法：

1. 单击 Debug 菜单中的 QuickWatch(SHIFT+F9)选项(参见图 6-28)。

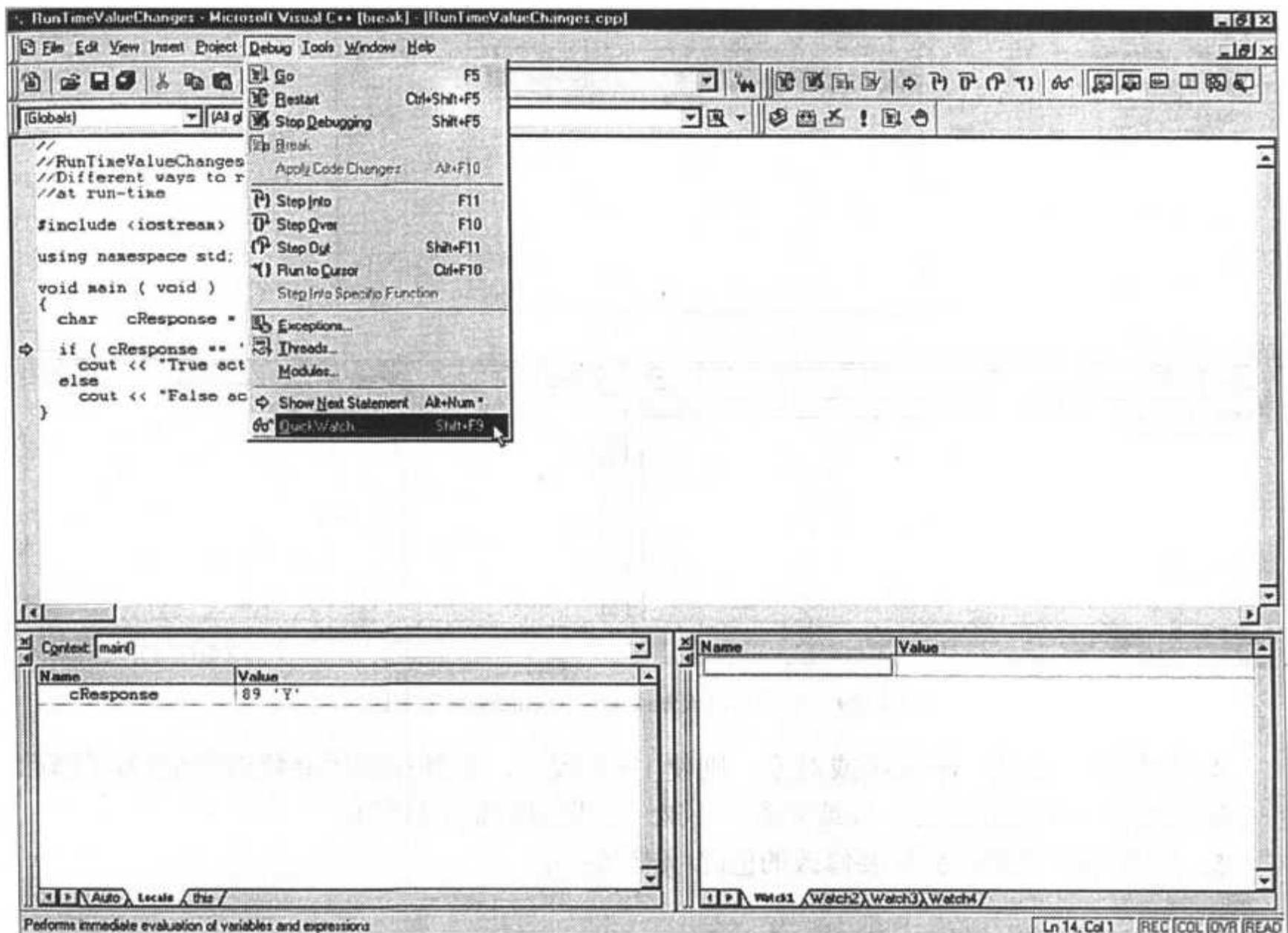


图 6-28 使用 QuickWatch 重新设置一个变量的值

2. 在 Expression 文本框中输入变量或寄存器的名字。
3. 单击 Recalculate 按钮(参见图 6-29)。

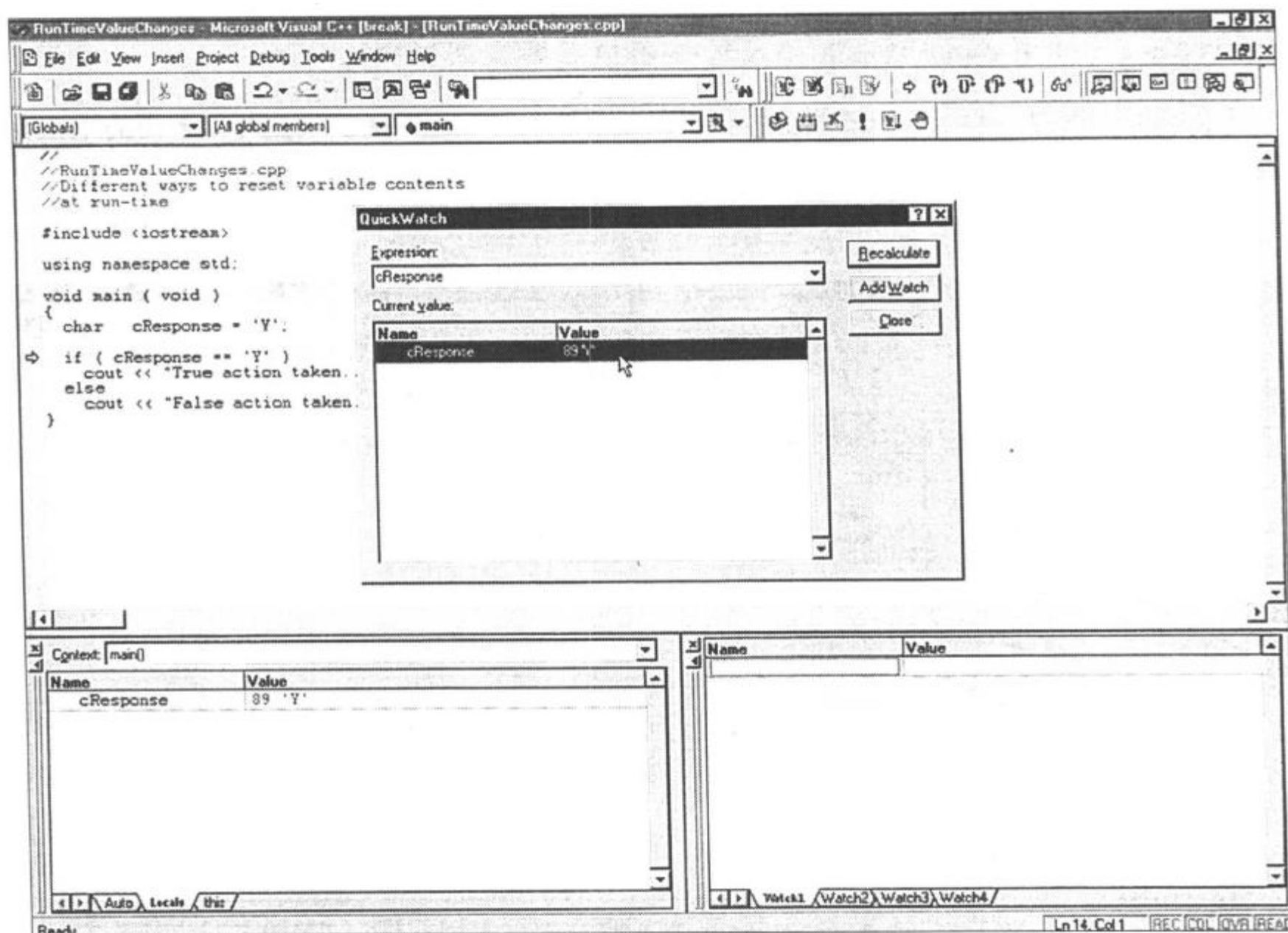


图 6-29 单击 QuickWatch 的 Recalculate 按钮

4. 如果该变量是一个数组或对象，则使用+框展开，直到找到所要修改的值(为了修改一个数组的值，必须修改各字段或元素，不能一次性编辑整个数组)。
5. 使用 TAB 键移动到所要修改的值(参见图 6-30)。

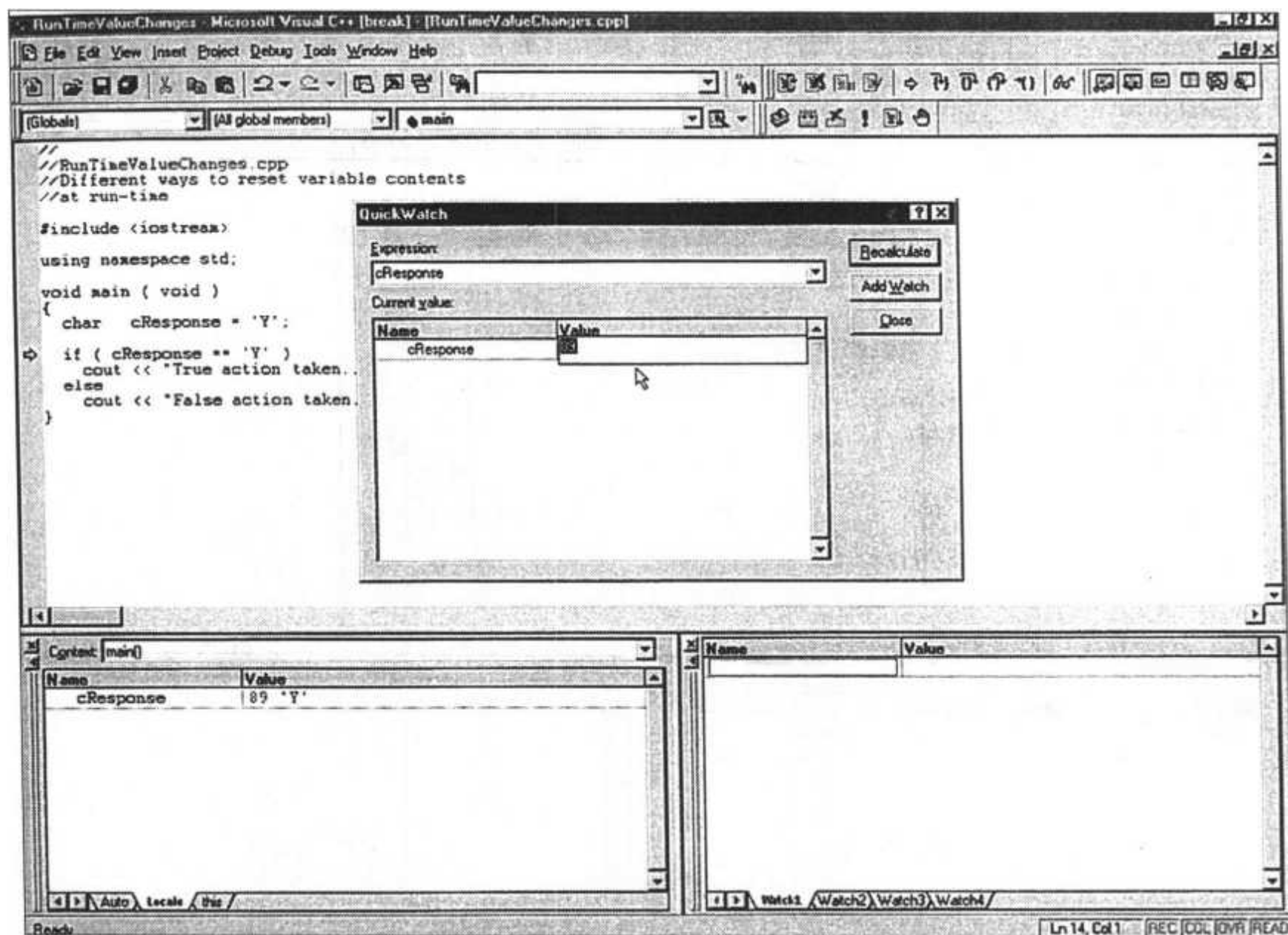


图 6-30 使用 TAB 移动到 Value 列

6. 输入新值后按 ENTER(参见图 6-31)。

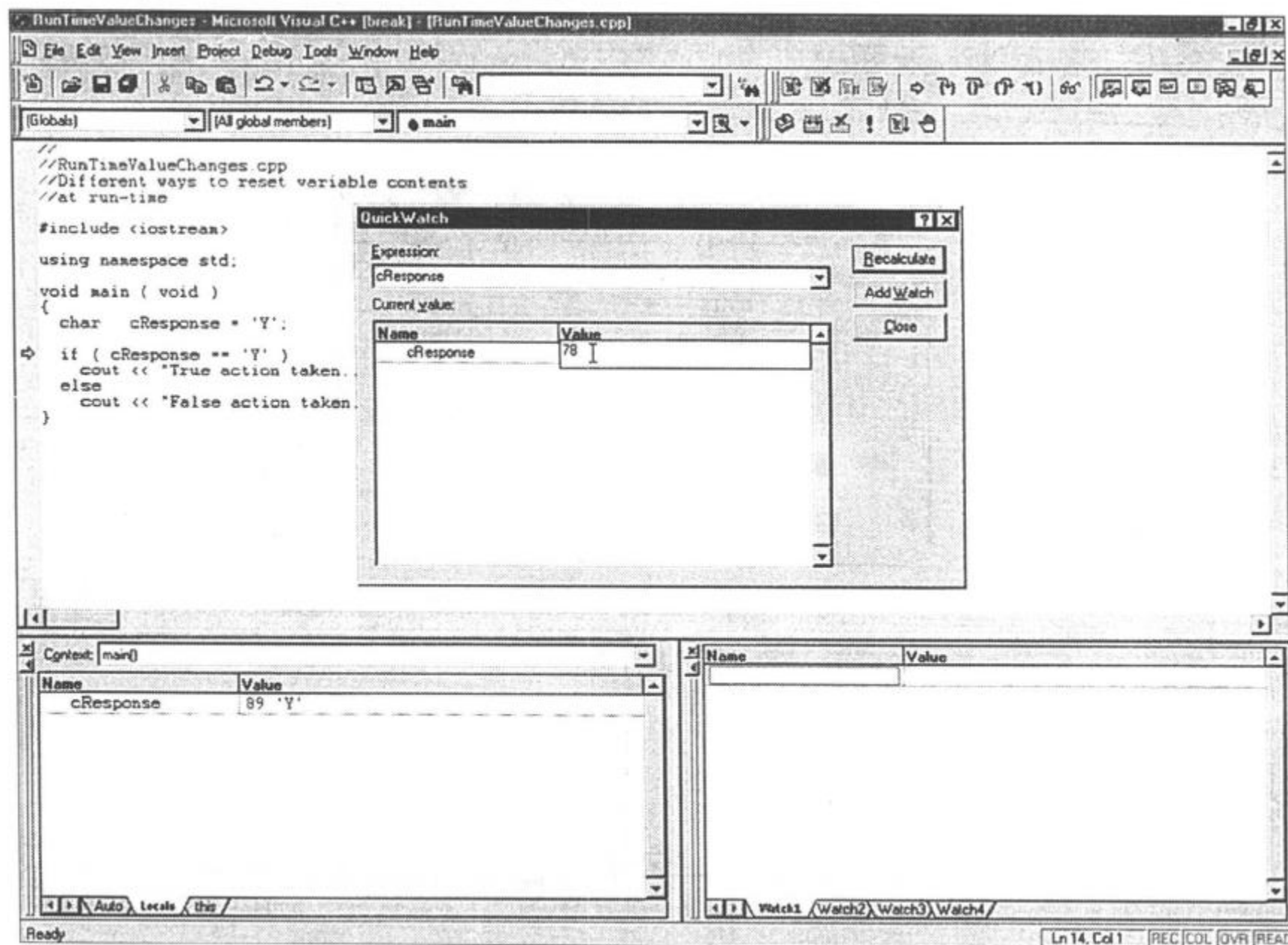


图 6-31 输入变量的新值

7. 单击 Close。

从图 6-32 中我们又一次看出，Debugger 现在已经做好准备，使用 `cResponse` 的“N”值单步运行 `if...else` 语句。

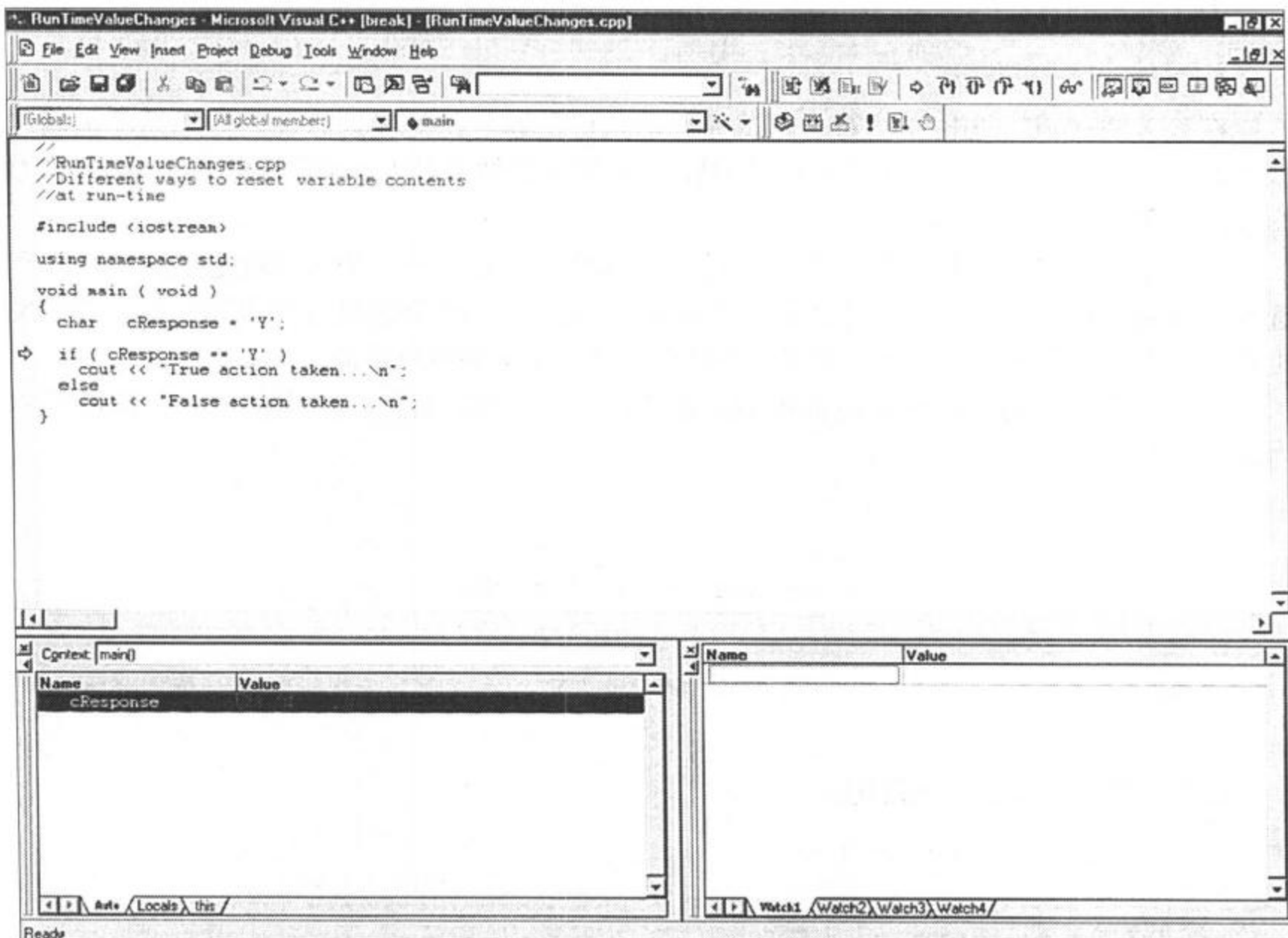


图 6-32 使用 cResponse 等于 “N” 调试

6.2.2 循环调试技巧

所有的软件工程师都知道，测试一个循环的迭代次数与设计的可靠和健壮程度是等价的。Microsoft Visual C++ Debugger 的建立可以有效地跟踪 **for**，**while** 或 **do...while** 循环体的循环迭代次数，这些循环的循环体中包含大量的代码语句、函数调用以及类方法调用。对于只检查循环迭代的次数而不关心循环的具体内容来说，即使重复使用 **Step Over(F10)** 也将效率很低。

然而，在演示聪明地设置断点的方法之前，首先考虑在准备调试一个 **for** 循环的迭代次数时只修改 **MAX_EXTENT** 的方法，比较如下的语句：

```
#define MAX_EXTENT 10000
for ( int iOffset = 0; iOffset < MAX_EXTENT; iOffset++);
```



为了测试的目的：

```
#define MAX_EXTENT 5
```

在许多情况下，这种方法对于测试循环迭代已足够。此处我们假设，如果循环正确迭代了 5 次，则其也应该迭代 10000 次。但是，这种简单的技术对于许多严重依赖于复杂的 I/O 交互的应用程序是不适合的。

当一个循环测试条件不适合前面所讨论的这种方法是，另一个选择就是在循环体内有选择地设置断点。从前面的讨论知道，**Debug|Go** 命令(F5)将全速执行程序到结束，或如果设置了断点，则全速执行算法直至下一个断点，然后停下等待单步调试。

循环体内断点的设置位置是由我们决定的，一般来说，有两个关键的位置，即循环体内的第一条语句：

```
while ( test_condition )
{
    statement1; // option 1:place breakpoint here
    statement2;
    //...
    statementn;
}
```

或循环体内的最后一条语句：

```
while ( test_condition )
{
    statement1;
    statement2;
    //...
    statementn; // option 2:place breakpoint here
}
```

当选择二者之一设置断点之后，只要运行(F5)程序，计算退出循环之前“遇到”断点的次数即可。每按 F5 一次(Debug|Run 命令)计一次。

6.2.3 调用调试函数

当从调试内联代码语句跳转到调试子程序调用时，出现逻辑错误的概率将迅速上升。如传送的值是否正确，所传送的变量是值调用还是引用调用，所需要的调用方法是否是所得到的等等。Microsoft Visual C++ Debugger 在 Call Stack 窗口中显示这些信息的功能是定位和诊断这些问题最有些的技术。

为了跟踪本节中的例子，读者需要输入如下的 `BadRecursiveCallStack.cpp` 程序：



```
//
//DebuggingSubroutines.cpp
//
#include <iostream>
using namespace std;
//Notice clean alignment of formal
//argument types...
int aSimpleSubroutine( char  cValue,
                      int    iValue,
                      float  fValue );

void main ( void )
{
    char  cValue = 'A';
    int    iValue = 125;
    float  fValue = 1.2345;

    aSimpleSubroutine ( cValue,
                      iValue,
                      fValue );
}
int aSimpleSubroutine( char  cValue,
                      int    iValue,
                      float  fValue )
{
    //subroutine body
    return iValue;
}
```

现在，我们需要学习的是如何调试子程序调用，所以不要按 F10 选择 Step Over 模式。而是按 F11，选择 Step Into 模式，直到屏幕的显示与图 6-33 一样。切记，Step Over 和 Step Into(F10 或 F11)执行完全相同的单步操作，除非跟踪箭头所在的位置为子程序调用语句。

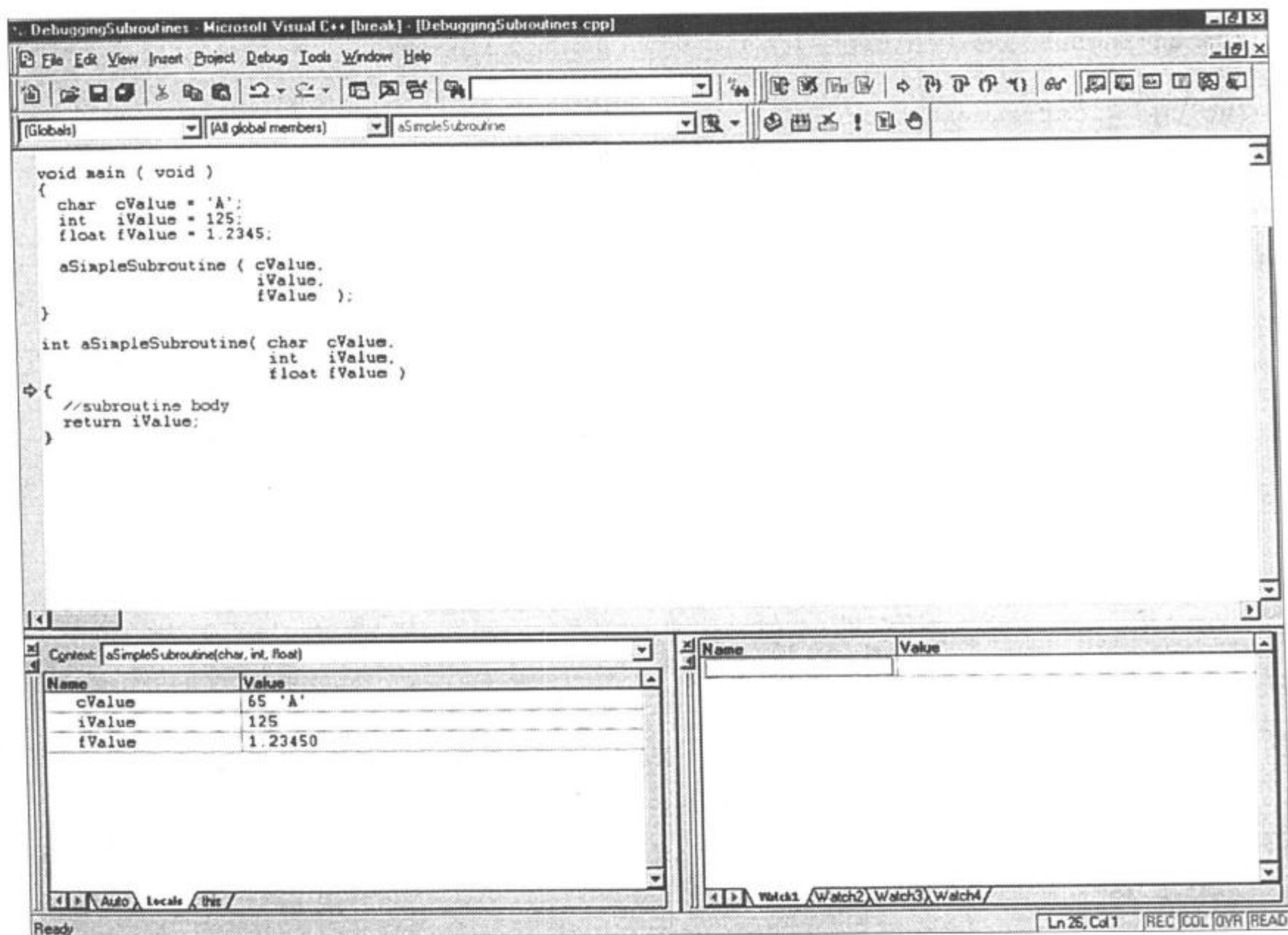


图 6-33 Step Into(F11) *aSimpleSubroutine* 跟踪显示初始的参数值

通过单击 Visual C++ View 菜单，选择 Debug Windows 选项，再选择 Call Stack(参见图 6-34)，可以打开 Visual C++ Call Stack 窗口。

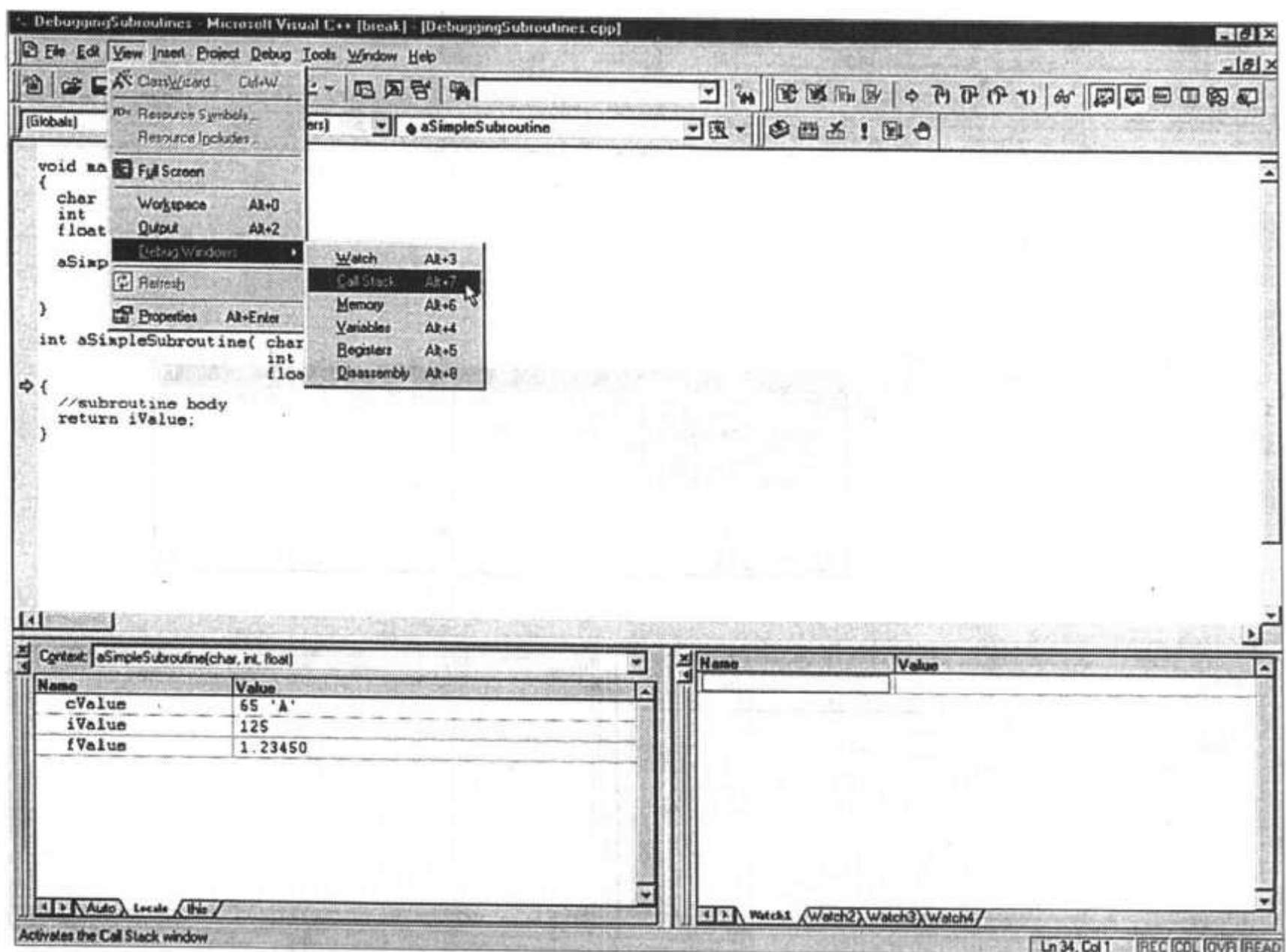


图 6-34 访问 Debug Windows|Call Stack(ALT+7)

如果读者跟随我们操作，则现在按 **ALT+7**，Call Stack 窗口出现，如图 6-35 所示。

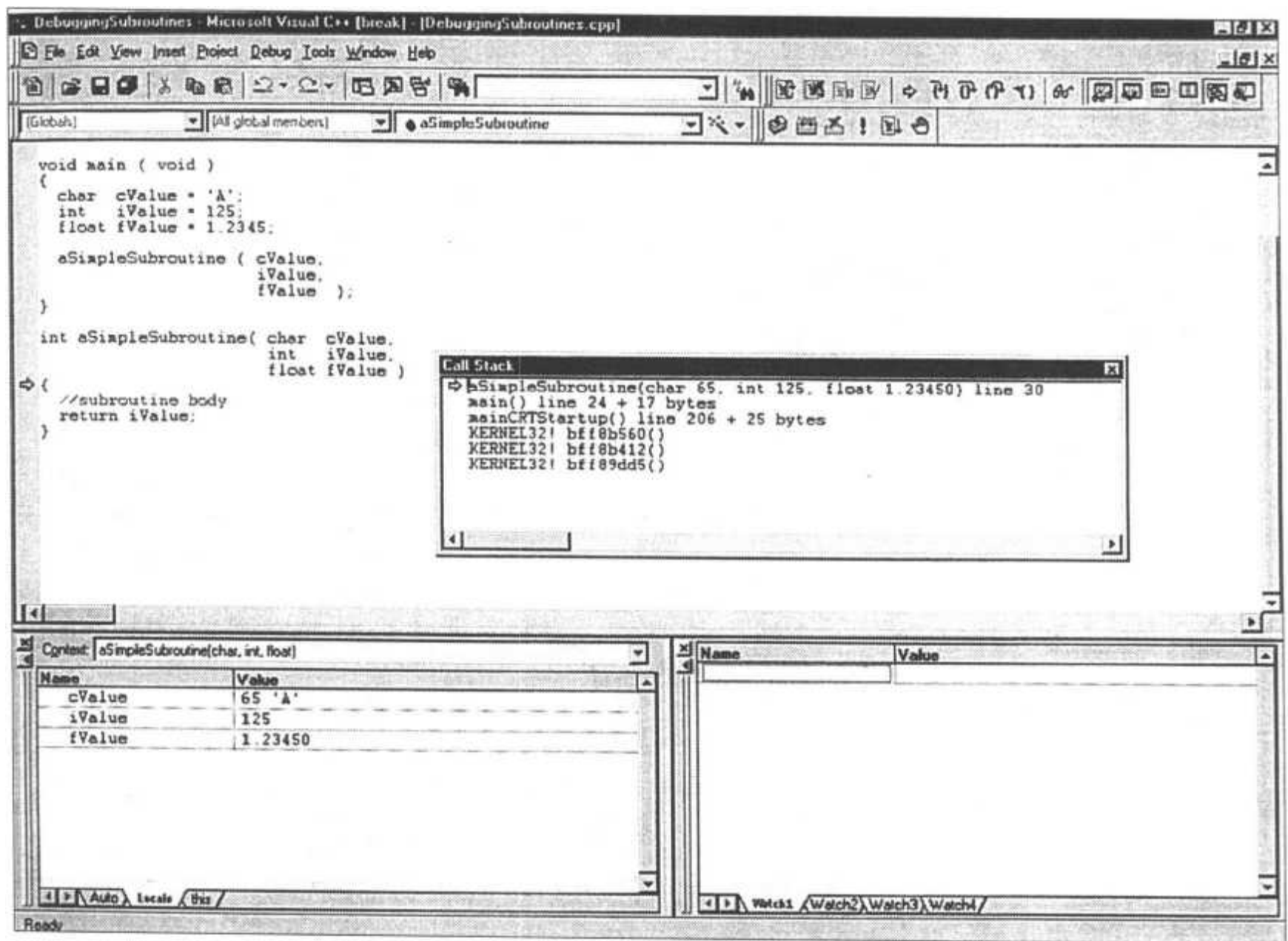


图 6-35 观察调用堆栈

可以看出，在 Call Stack 窗口的左边有一个小的黄色箭头(缺省颜色)指向 *aSimpleSubroutine*，这是(在这一例子中)最后一个被调用子程序(函数或类方法)的函数名。

Call Stack 窗口显示该子程序的名字，并在其后给出每一个参数的数据类型和当前值，如果有的话。在函数名下方，可以看到调用子程序的名字以及调用语句所在的行号和字节偏移量。图 6-35 中，在这一信息之后列出了 *mainCRTStartup()* 子程序、调用语句行号以及字节偏移量，然后是每一个子程序在 RAM 中所在位置的 KERNEL32 物理地址。

仅仅为了比较的目的，图 6-36 给出了同一个算法的另一种编写方法，该方法使用了按照引用调用参数的调用规范，而不是图 6-35 中所跟踪的按照值调用的方法。注意 Call Stack 窗口是如何标记参数传递调用规范的这种变化的，它在形式参数的数据类型中包含了地址操作符 *&*。图 6-35 中的参数类型和值 *char 65, int 125, float 1.23450*，在图 6-36 中替换为 *char & 65, int & 125, float & 1.23450*，表示调用规范为按引用调用。

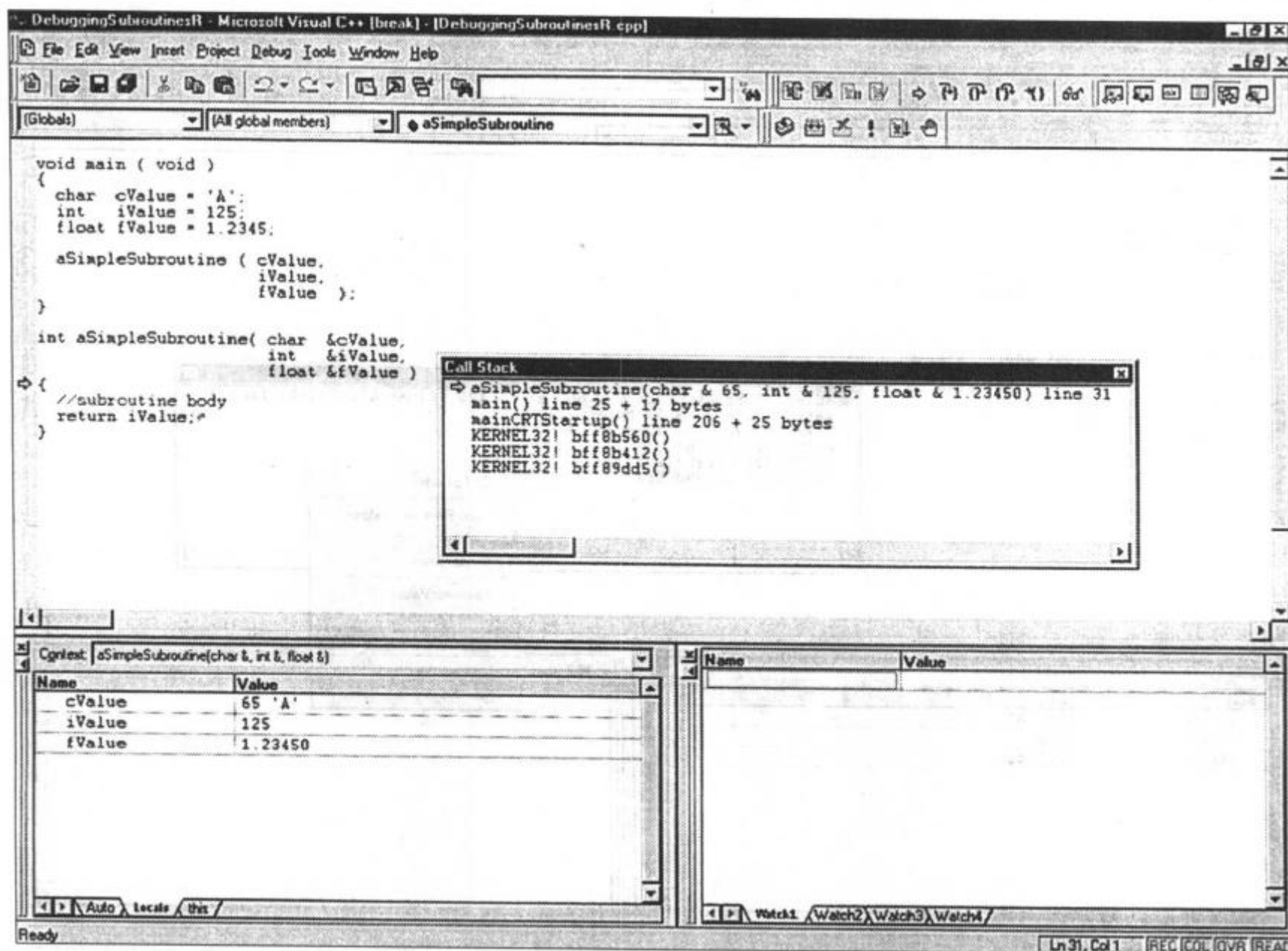


图 6-36 使用按引用调用规范重新编写程序

用户可以定义 Call Stack 窗口内所显示的信息类型。通过右击 Call Stack 窗口(参见图 6-37)，可以打开/关闭 Parameter Values 和 Parameter Types，然后在缺省的输出值精度与 Hexadecimal Display 格式之间切换。

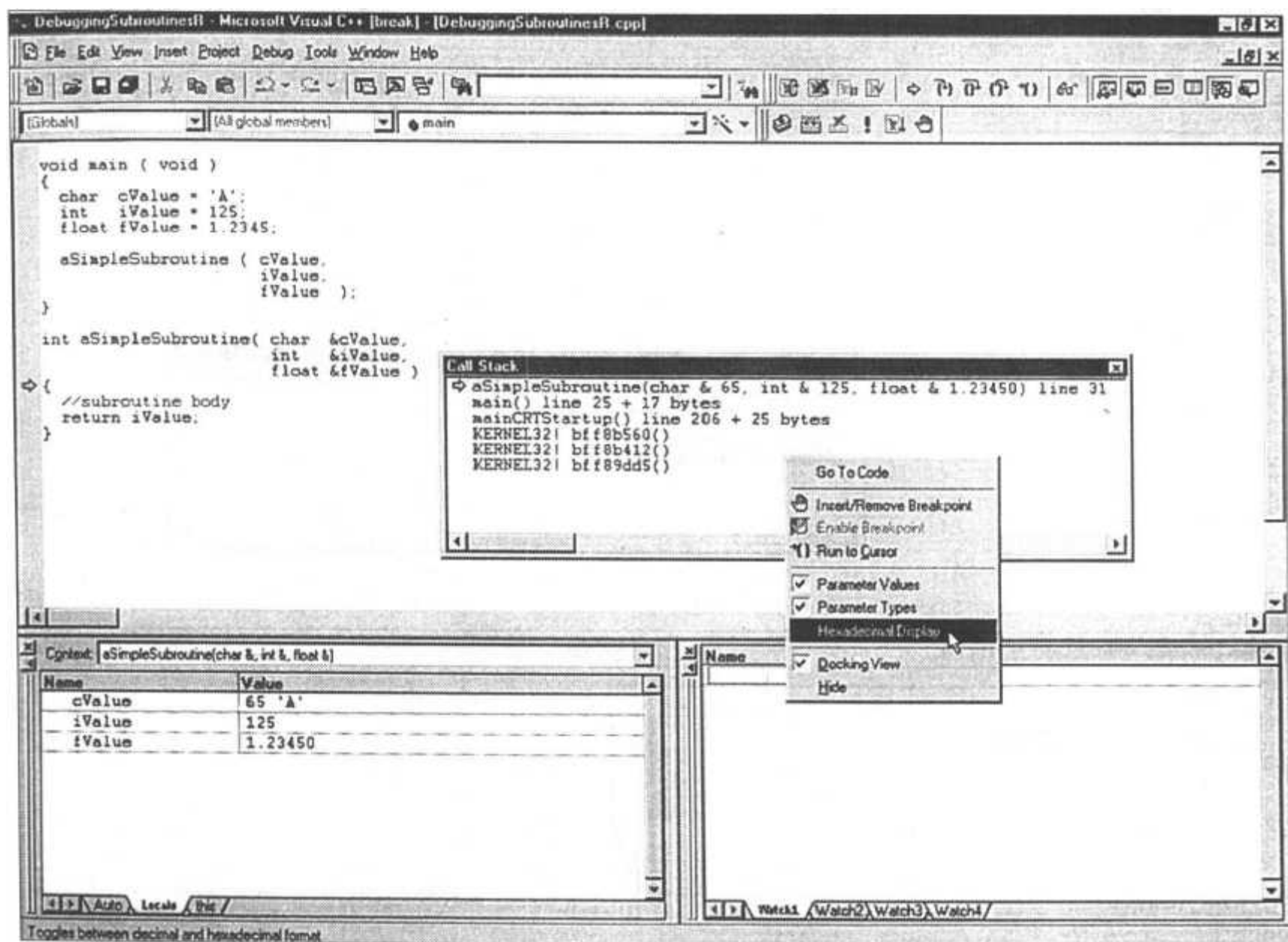


图 6-37 Call Stack 本地菜单(在 Call Stack 窗口内右击)选项

Call Stack 窗口还有一个更有用的特性。通过双击调用过程(在本例中为 **main()**)的名字，Visual C++ Debugger 将自动定位源文件和调用语句。两个淡绿色的箭头突出显现了 Call Stack 窗口与自动跟踪源代码语句之间的关系(参见图 6-38)。

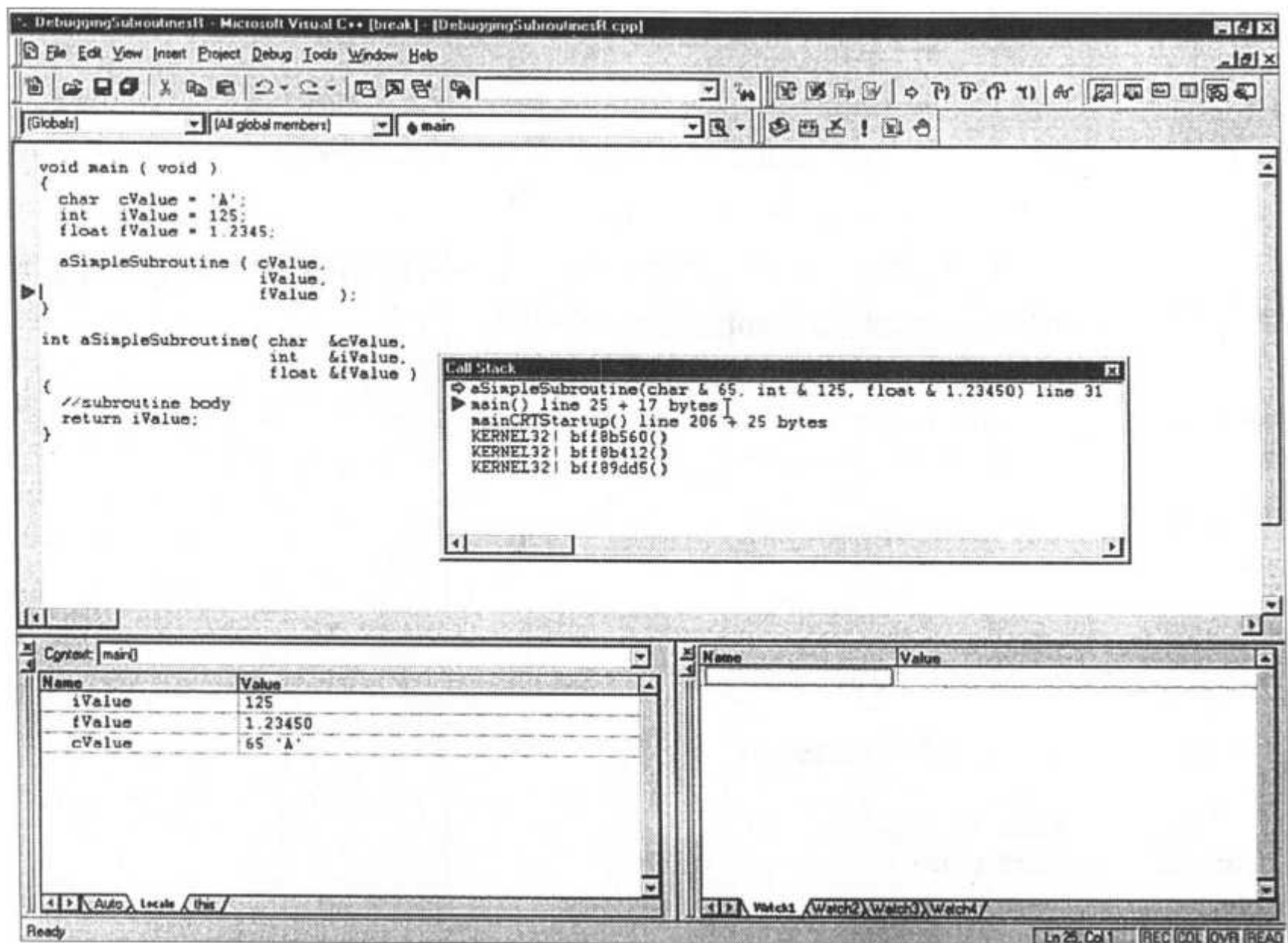


图 6-38 定位调用程序与调用语句

6.2.4 递归调用与调用堆栈

许多计算机学老师对于递归所给出的定义真是光怪陆离，例如，“递归定义——见递归。”当读者第一次在其正规培训中遇到递归算法时，毫无疑问将感到那些定义完全匹配了自己头脑中的含混。现在看来实际上是一个程序调用其自己。对于这样一种古怪的特性有什么可能的用处？随后学习了二叉树，In-Fix, Pre-Fix 和 Post-Fix 算法以及树的遍历。

虽然递归算法的确简明、紧凑并且有着迷人的代码段，但要跟踪和调试递归算法则需要有一定的专门知识和经验。Microsoft Visual C++ Debugger 的 Call Stack 窗口正是我们需要的，在这些聪明的代码重用程序中检测错误的工具。

例如，请看下面的代码段。在我们的软件工程生涯中，如果看到过多个“Hello World!”程序，或者多个递归阶乘实例的话，那么我们最好作一个旅行代理。然而，递归阶乘算法确实是一个理想的起点。该算法本身非常简单，只要从一个初始值开始，例如该初始值为 4，



乘以比 4 小 1 的数 3，乘以比 3 小 1 的数 2，依此类推直到到达 1 为止：

$4 * 3 * 2 * 1$

直接编写这样的算法必定不会反应反映出程序员的专业水平：

```
iRecursiveCalculation = 4 * 3 * 2 * 1;
```

所以很自然地，我们选择了一种通用的递归算法，如下面的应用程序所示(如果读者要跟踪，则输入 **BadRecursiveCallStack.cpp** 程序)：153-154

```
//  
//BadRecursiveCallStack.cpp  
//Program demonstrates recursive calls  
//  
int Factorial( int iCurrentValue );  
void main ( void )  
{  
    int iRecursiveCalculation;  
    iRecursiveCalculation = Factorial( 4 );  
}  
int Factorial( int iCurrentValue )  
{  
    if( iCurrentValue == 1 )  
        return iCurrentValue;  
    else  
        return iCurrentValue *=  
            Factorial( iCurrentValue -- );  
}
```

非常高兴，我们开始了。试设想一下烦琐的 Debugger 跟踪将如何，仅仅为了找出所计算的 `iRecursiveCalculation` 值都是不可能的。幸运的是，我们知道 Debugger 的 Call Stack 窗口，并且开始一次次地调用检查。图 6-39 如我们希望的那样开始了(使用我们知道的关于 **Step Over** 和 **Step Into** 的知识，改变 Debugger 的显示使其类似于图 6-39 所示)。

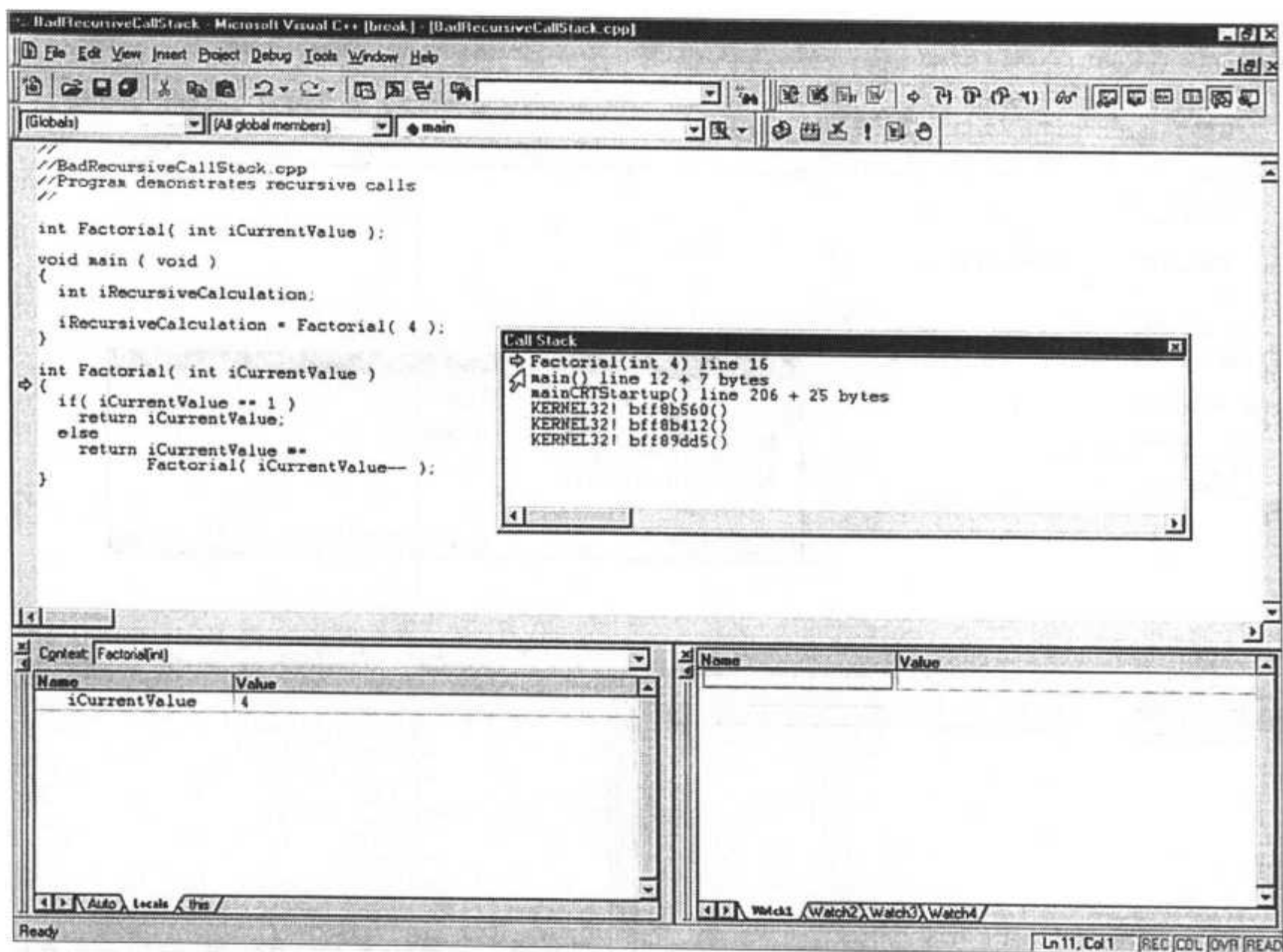


图 6-39 初始递归 Factorial 调用显示 Call Stack 栈顶的值为 4

即使对 `Factorial()` 的第二次调用，看起来并不十分糟糕，参见图 6-40(使用我们知道的关于 **Step Over** 和 **Step Into** 的知识，改变 Debugger 的显示使其类似于图 6-40 所示)。

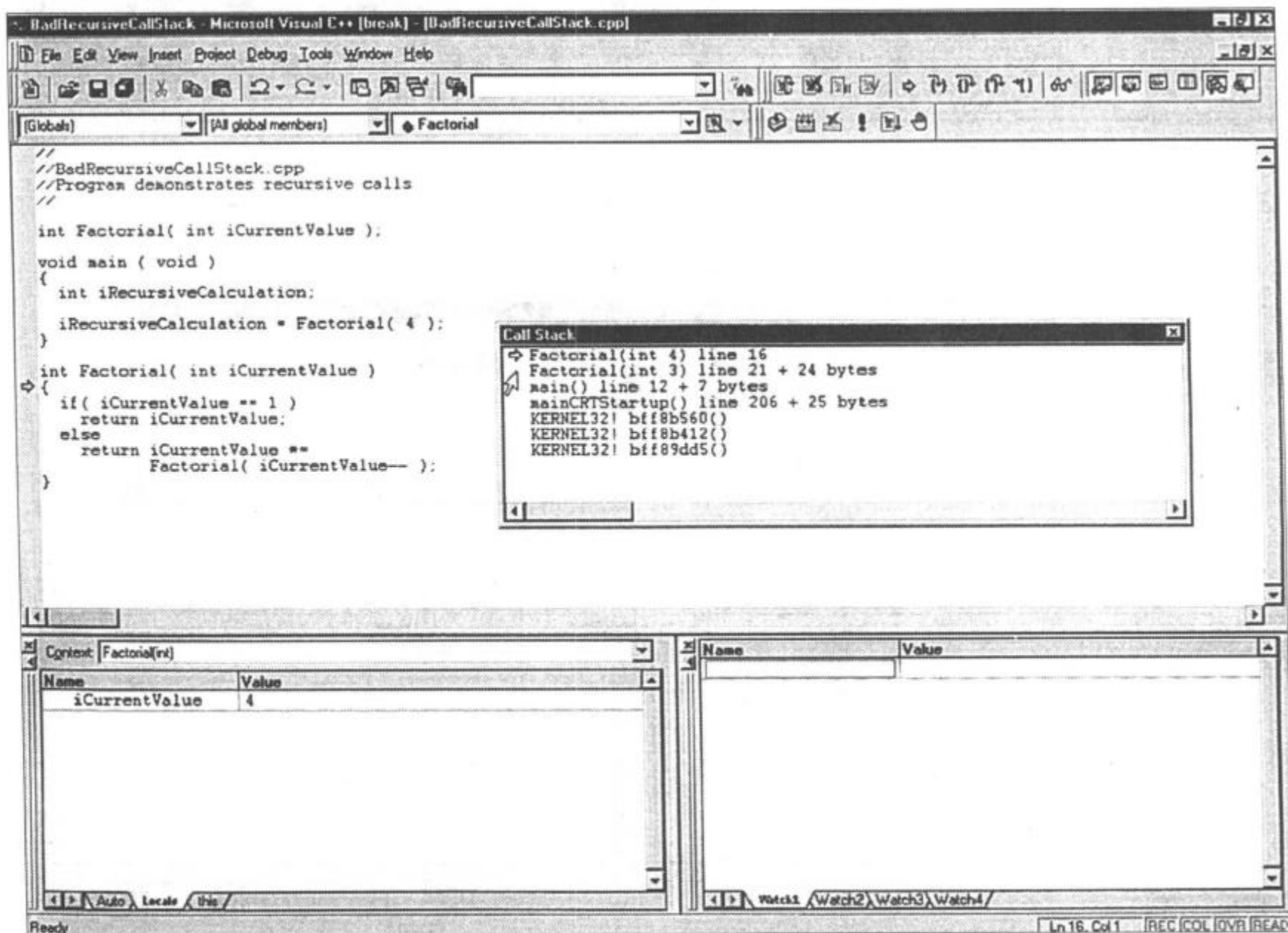


图 6-40 第二次调用递归 Factorial 子程序

然而，在第三次跟踪进入该函数时，立刻可发现一个严重的计算错误开始传播(使用我们知道的关于 Step Over 和 Step Into 的知识,改变 Debugger 的显示使其类似于图 6-41 所示)。

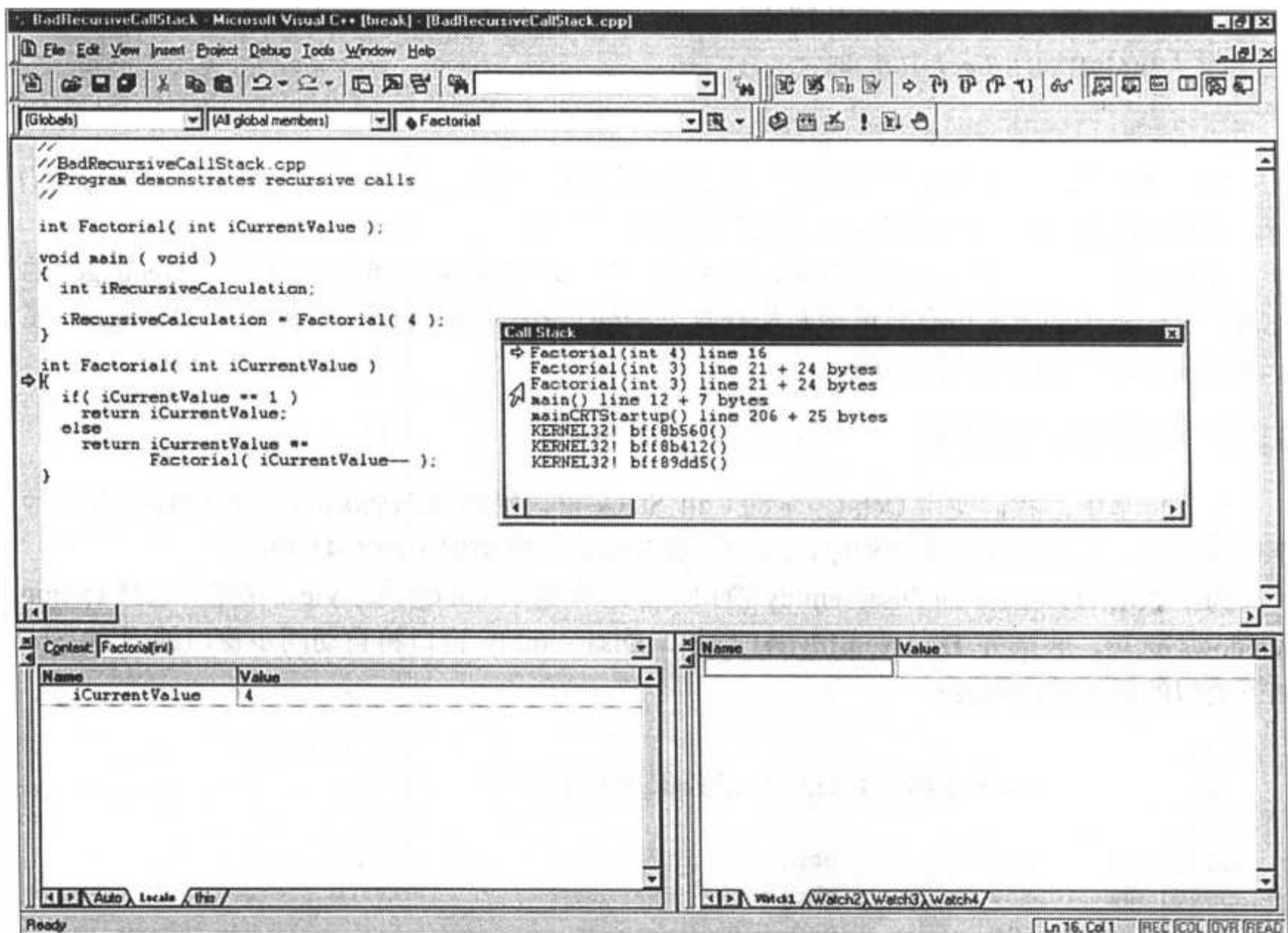


图 6-41 认识到 Factorial 的计算错误已经开始

如果该算法正常运行，则应该看到每一次调用该函数 *Factorial()* 时，所传送的值比前一次调用的值小 1。显然，在第三次跟踪进入时，Call Stack 窗口标志出了一个错误。

现在，如果读者立即检测到 *Factorial()* 代码中的错误，那么应该出去买点什么奖励一下自己。然而，没有多年与 C/C++ 的摸爬滚打，代码看起来似乎还是没有问题。

产生计算错误的恼人的代码语句涉及到了 *Factorial()* 的递归调用语句(倾斜加粗是为了强调):

```
int Factorial( int iCurrentValue )
{
    if ( iCurrentValue == 1 )
        return iCurrentValue;
    else
        return iCurrentValue * =
```



```
Factorial( iCurrentValue-- );
```

当从 `main()` 调用 `Factorial()` 时:

```
iRecursiveCalculation = Factorial ( 4 );
```

对于所传送值的解释不存在问题, 此处该值为 4。然而当从 `Factorial()` 中调用 `Factorial()` 时, 所传送的值是 `iCurrentValue--`。仍然不清楚, 是否?

此时, 程序员可能砍掉这一方案, 或者好一些, 使用 Microsoft Visual C++ Debugger 的工具。这一复杂的方法可有效地诊断和修复 `Factorial()` 的代码错误, 涉及到了 Debugger 显示源代码段的反汇编版本的功能。

6.2.5 查看反汇编代码

在上一节中, 我们使用 Debugger 的 Call Stack 窗口检测到 `Factorial()` 子程序体中存在一个计算错误。如果执行本章的程序, 监视器屏幕应该仍然类似于图 6-41 所示。

为了激活 Debugger 的 Disassembly 窗口, 首先单击 Visual C++ 的 View 菜单, 选择 Debug Windows 选项, 再选择 **Disassembly(ALT+8)**。Disassembly 窗口将自动同步窗口的内容为与如下类似的反汇编代码段:

```
14:
15:      int Factorial ( int iCurrentValue )
16:      {
00401070      push      ebp
00401071      mov       ebp, esp
00401073      sub       esp, 44h
00401076      push      ebx
00401077      push      esi
00401078      push      edi
00401079      lea       edi, [ebp-44h]
0040107C      mov       ecx, 11h
00401081      mov       eax, 0CCCCCCCCh
00401086      rep stos  dword ptr [edi]
17:      if( iCurrentValue == 1 )
00401088      cmp       dword ptr [ebp+8], 1
0040108C      jne       Factorial+23h (00401093)
18:      return iCurrentValue;
0040108E      mov       eax, dword ptr [ebp+8]
00401091      jmp       Factorial+4Ah (004010ba)
19:      else
20:          return iCurrentValue *=
21:          Factorial( iCurrentValue-- );
```



```

00401093    mov     eax,dword ptr [ebp+8]
00401096    mov     dword ptr [ebp-4],eax
00401099    mov     ecx,dword ptr [ebp-4]
0040109C    push    ecx
0040109D    mov     edx,dword ptr [ebp+8]
004010A0    sub     edx,1
004010A3    mov     dword ptr [ebp+8],edx
004010A6    call    @ILT+0(Factorial) (00401005)
004010AB    add     esp,4
004010AE    mov     ecx,dword ptr [ebp+8]
004010B1    imul    ecx,eax
004010B4    mov     dword ptr [ebp+8],ecx
004010B7    mov     eax,dword ptr [ebp+8]
22:    }
    
```

当学习了 Disassembly 窗口的内容后，需要重新编辑 **Factorial()** 的错误语句，将其从：

```

int Factorial( int iCurrentValue )
{
    if ( iCurrentValue == 1 )
        return iCurrentValue;
    else
        return iCurrentValue *
            Factorial( iCurrentValue-- );
    
```

改变为：

```

int Factorial( int iCurrentValue )
{
    if ( iCurrentValue == 1 )
        return iCurrentValue;
    else
        return iCurrentValue *
            Factorial( iCurrentValue - 1 );
    
```

再次执行编译连接并开始一个新的跟踪，一定要保证调用 **Factorial()** 至少三次。现在 Debugger 窗口应类似图 6-42 所示。

现在，打开 Disassembly 窗口。

```

14:
15:    int Factorial ( int iCurrentValue )
16:    {
00401070    push    ebp
00401071    mov     ebp,esp
00401073    sub     esp,40h
    
```



```
00401076    push    ebx
00401077    push    esi
00401078    push    edi
00401079    lea     edi,[ebp-40h]
0040107C    mov     ecx,11h
00401081    mov     eax,0CCCCCCCCh
00401086    rep stos dword ptr [edi]
17:        if( iCurrentValue == 1 )
00401088    cmp     dword ptr [ebp+8],1
0040108C    jne     Factorial+23h (00401093)
18:        return iCurrentValue;
0040108E    mov     eax,dword ptr [ebp+8]
00401091    jmp     Factorial+4Ah (004010bae)
19:        else
```

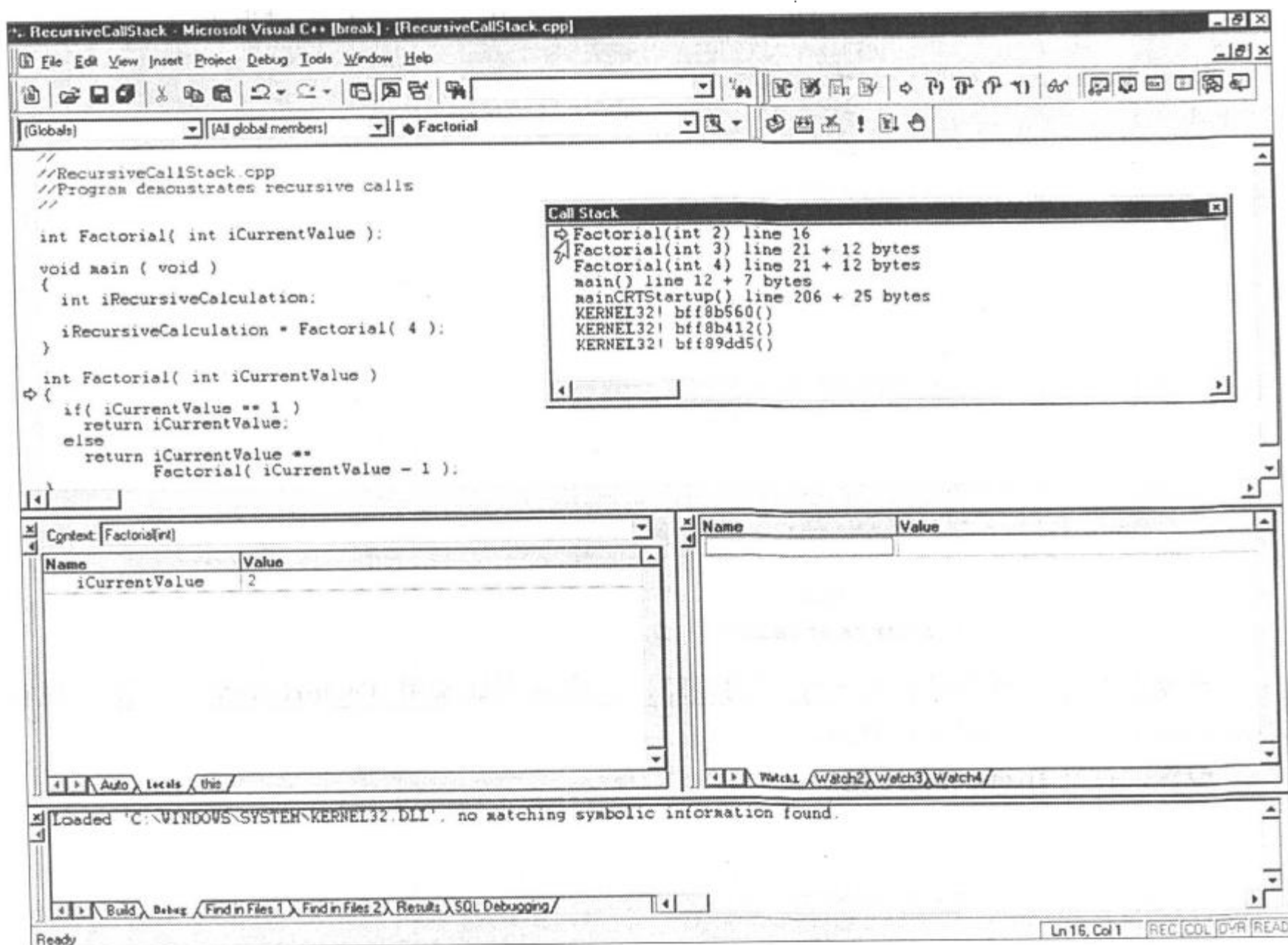


图 6-42 显示 Factorial()正确的 Call Stack 值



```

20:      return iCurrentValue *=
21:      Factorial( iCurrentValue - 1 );
00401093  mov     eax,dword ptr [ebp+8]
00401096  sub     eax,1
00401099  push    ecx
0040109A  call    @ILT+0(Factorial) (00401005)
0040109F  add     esp,4
004010A2  mov     ecx,dword ptr [ebp+8]
004010A5  imul    ecx,eax
004010A8  mov     dword ptr [ebp+8],ecx
004010AB  mov     eax,dword ptr [ebp+8]
22:      )
    
```

表 6-1 中比较了两个最重要的代码段。左边的一列给出了 Disassembly 窗口所显示的 *iCurrentValue*-的翻译本，而右边的一列给出了重新编写的公式 *iCurrentValue - 1* 的翻译版本。

表 6-1 比较两个 *iCurrentValue* 公式的反汇编翻译

<i>iCurrentValue</i> --			<i>iCurrentValue - 1</i>		
21:		Factorial(<i>iCurrentValue</i> --);	21:		Factorial(<i>iCurrentValue</i> --);
00401093	mov	eax,dword ptr [ebp+8]	00401093	mov	eax,dword ptr [ebp+8]
00401096	mov	dword ptr [ebp+4],eax	00401096	sub	eax,1
00401099	mov	ecx,dword ptr [ebp+4]	00401099	push	eax
0040109C	push	ecx	0040109A	call	@ILT+0(Factorial)
0040109D	mov	edx,dword ptr [ebp+8]	(00401005)		
004010A0	sub	edx,1			
004010A3	mov	dword ptr [ebp+8],edx			
004010A6	call	@ILT+0(Factorial)			
(00401005)					

对照的反汇编代码段清楚地反应出了第一种算法中不正确的值的位置。第一列中 *ecx* 寄存器的引用表示减量操作符的汇编语言对等物。此处，原始 *iCurrentValue* 的值再次被使用，这样产生了错误。右边的列中只引用了 *eax* 寄存器，在传送新值到下一个递归子程序调用之前，已经建立了正确的值。

注意 Call Stack 窗口中所报告错误值是如何有机结合的，正式折衷结合使得我们发现了算法中的错误所在，并且对 Disassembly 窗口内容的技术性使用解释了为什么该算法不正确。

在所有后面的各章中包含这样的例子，这些例子说明机械的 Debugger 技巧如何使用 Step Into 模式，与逻辑布置或调试概念结合起来，提高了 Debugger 的性能。

6.3 进一步观察变量

当调试一个应用程序时，Variables 窗口及其自动跟踪范围之内变量的功能，正是我们最



初检测一条有错误的代码语句时所需要的。然而随着应用程序的越来越复杂——从一个简单的过程到一个面向对象功能全面的多线程 Windows 应用程序——变量之间的交互也不断地增加。在这种情况下，我们将需要使用组合观察。

下面的两节将进一步了解 Debugger 的 QuickWatch 和 Watch 窗口。在以后各章的适当位置，所有三个窗口：Variables、QuickWatch 和 Watch 都将得到更复杂的使用。

6.3.1 使用 QuickWatch 窗口

可以使用 QuickWatch 快速检查 SQL 变量和参数的值，也可以使用 QuickWatch 修改局部变量的值或添加变量到 Watch 窗口中。但不能修改全局变量的值。

QuickWatch 对话框包含一个文本框，可以在该文本框中输入变量的名字和一个电子表字段，显示所指定变量的当前值。Current Value 电子表字段一次只显示一个变量。在文本框中输入一个新的变量后按 ENTER 键，可替换原先的变量。QuickWatch 以其内部格式显示 SQL 代码变量的值。

说明

Microsoft 文档中说明只有企业版可以调试 SQL 源代码。

6.3.1.1 使用 QuickWatch 观察变量(主菜单方法)

为了使用 QuickWatch，必须首先启动 Debugger，并在激活一个有意义的 QuickWatch 窗口之前，执行通过一些变量声明或代码段。为启动 QuickWatch，可以：

1. 单击 Debug 菜单，再单击 QuickWatch，QuickWatch 对话框出现。
2. 输入或粘贴变量名到 Expression 文本框。

或：

从 Expression 下拉列表框中单击一个变量名，该列表框中包含最新使用过的 QuickWatch 标识符。

3. 单击 Recalculate。
4. 单击 Close。

6.3.1.2 使用 QuickWatch 观察变量(快速方法)

另一种启动 QuickWatch 的方法是，首先等待到 Debugger 停止，切换到源代码窗口，右击一个变量，然后：

1. 从弹出的快捷菜单中单击 QuickWatch。如果 I 光标处于源代码窗口中一个可以观察的变量上，QuickWatch 将自动输入该变量的名字到 Expression 下拉列表框中。
2. 单击 Recalculate。



3. 单击 Close。

6.3.1.3 使用 QuickWatch 修改一个变量的内容

当程序暂停于一个断点或两步之间时，可以修改程序中的任何一个变量。这一功能为试验各种变化并实时查看其结果，或从某逻辑错误中得到恢复提供了很大的灵活性。为了使用 QuickWatch 修改一个变量的内容，可以：

1. 选择 Debug|QuickWatch 选项。
2. 在 Expression 文本框中输入变量名。
3. 单击 Recalculate。
4. 使用 TAB 键移动到需要修改的值。
5. 输入新值后按 ENTER 键。
6. 单击 Close。

6.3.1.4 添加一个 QuickWatch 变量到 Watch 窗口

通过单击 QuickWatch 窗口的 Add Watch 按钮，可以很容易地将一个 QuickWatch 变量永久性地传送到 Watch 窗口中。

6.3.2 使用 Watch 窗口

切记，Watch 窗口中的内容与 Variables 或 QuickWatch 窗口中的内容不同，Watch 窗口中的内容是静态的，而不是动态的。在调试一个程序时，可以使用 Watch 窗口指定所要观察的局部或全局变量。也可以使用该窗口修改一个局部变量的值，但不能修改全局变量的值。

6.3.2.1 添加变量到 Watch 窗口

Watch 窗口中包含四个标签：Watch1、Watch2、Watch3 和 Watch4，每一个标签都在一个电子表字段中显示一个用户指定的变量列表，可以将需要观察的变量分组放在同一个标签中。例如，可以将与一个特定窗口相关的变量放在一个标签中，而将与一个对话框相关的变量放在另一个标签中。这样，在调试该窗口时可能需要观察第一个标签，而在调试该对话框时需要观察第二个标签。为添加一个变量到 Watch 窗口，可以：

1. 单击 View|Debug Windows|Watch 选项。
2. 为变量选择一个标签(缺省为 Watch1)。
3. 输入、粘贴或拖动一个变量名到该标签的 Name 列。
4. 按 ENTER 键。

5. Watch 窗口立即求出该变量的值，并显示该值或错误消息。Watch 窗口以其内部 SQL 格式显示值。



6.3.2.2 修改 Watch 窗口显示格式

Watch 窗口不显示变量类型信息，但可以通过使用窗口的 Properties 页显示变量的类型信息。为了查看一个变量的类型信息，选择包含需要查看其类型的变量的行，然后右击 Watch 窗口，从快捷菜单中单击 Properties，或从 View 菜单中单击 Properties 即可。

6.3.2.3 从 Watch 窗口中删除变量

为从 Watch 窗口中删除一个变量，选择待删除变量所在的行后按 DELETE 键即可。当程序暂停在一个断点或两步之间时，可以修改程序中局部变量的值。这为试验各种变化并实时查看其结果，或从某个逻辑错误中得到恢复，提供了灵活性。

6.3.2.4 修改 Watch 窗口中变量的内容

为使用 Watch 窗口修改一个变量的值，可双击该变量的值，输入新值，然后按 ENTER 键即可。

错误监视

Microsoft Visual C++ Debugger 不允许修改图像、文本或日期数据类型。另外，也不能增加一个字符串变量(*szString*)中字符的个数。

6.4 小结

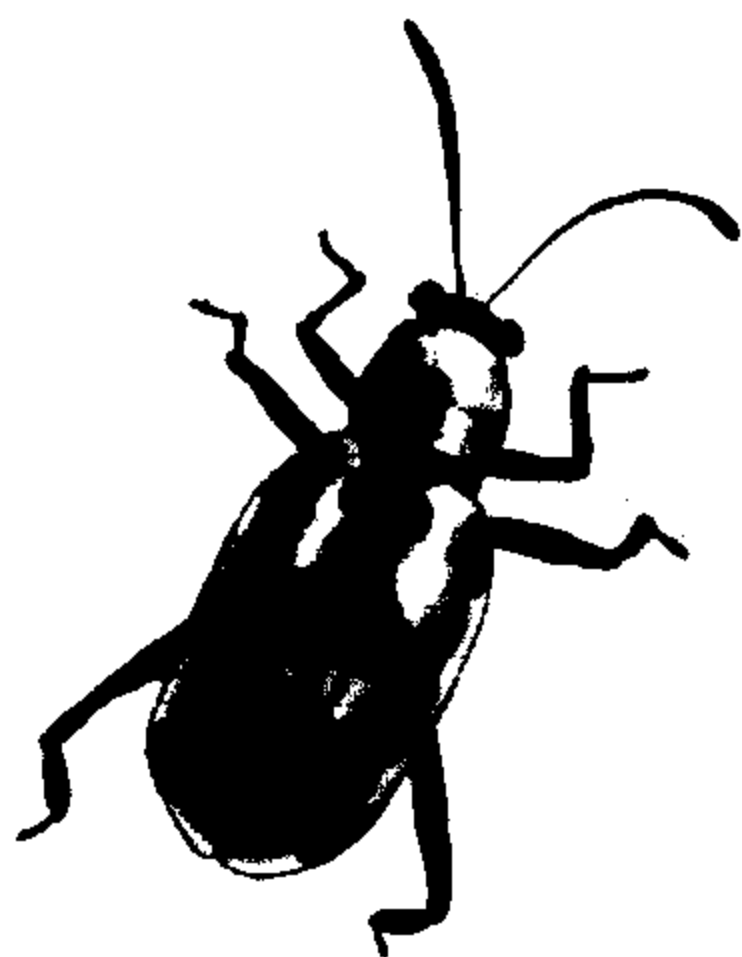
读者即使已经熟悉了标准 Debugger 的基本知识，如 Watch 窗口、Step Into、Step Over，也可能并不知道 Debugger 执行复杂任务的功能，如从当前位置使用新值调试。本章还提供了一些高级主题，如调用堆栈和通过灵活地插入断点有效使用标准 Debugger 的基本功能。随着以后的实践，读者将越来越有效地进行调试，并花费越来越少的时间调试“老的问题”。

下一章中，我们将学习 Debugger 与汇编语言代码相关的功能。这是程序员工具箱中一个激动人心的附件，因为从那时起，C/C++就与汇编语言代码有了同甘苦共命运的关系。



第 7 章

调试内联汇编语言代码





在 Microsoft Visual C++ 编译器中，提供了在一般 C++ 源代码工程中包含内联汇编语言代码的能力。近年来，由于 C 和 C++ 的增强，汇编语言代码的使用已经不再那么重要。然而，对于汇编语言的需求仍然存在，并且有可能在将来变成许多 C++ 工程的一部分。

在使用一个编译器时，我们实际上精确地受到了编译器开发者的支配。如果没有包含特殊的语言特性，那么些我们并不能为其添加这些功能。遗漏特性最好的例子是 IBM Pascal 编译器的第一个发行版本。该 Pascal 编译器的运行是正常的，但其没有包含任何处理图形的功能，即使是屏幕上的一个点，它都不能处理。一个给 IBM 的电话得到的回答是“是的，你必须编写一个汇编语言补丁处理图形。”真是奇妙的回答。

真实的情况是，计算机可以做的任何事情都可以通过汇编语言编程实现。这是任何编译语言都不敢吹嘘的事实。

本章将快速地复习汇编语言的基础知识，研究 Debugger 的功能，然后通过几个程序演示这些功能。这些汇编语言程序将演示这样的结果，使用汇编语言代码很容易完成，而如果不可能的话，使用 C++ 则很难完成。

7.1 汇编语言初步

本节并不是汇编语言图书或教程的替代物，其设计目的只是强调汇编语言的某些特性，这些特性将要在本章中演示。如果读者没有使用汇编语言编写过程序，则在继续阅读本章之前，应该首先寻找一本好的汇编语言图书看看。

7.1.1 数据类型

汇编语言程序员最经常使用的数据类型包括字节、字、双字及四字。表 7-1 给出了这些数据类型、表示法、位数及可以容纳它们的微处理器寄存器。

表 7-1 汇编语言数据类型

数据类型	表示法	长度(位)	可以使用的寄存器
字节	DB	8	al、ah、bl、bh、cl、ch、dl、dh
字	DW	16	ax、bx、cx、dx
双字	DD	32	eax、ebx、ecx、edx
四字	DQ	64	无

在汇编语言中，也可以寻址单个位，或使用 10 个字节的数据类型(DT)，其长度为 80 位。



7.1.2 寄存器

汇编语言所使用的寄存器实际上存在于微处理器自身内。从 80386 微处理器开始，多用途的寄存器包括 32 位的 `eax`、`ebx`、`ecx` 及 `edx` 寄存器。16 位寄存器实际上是 32 位父寄存器的一部分，包括 `ax`、`bx`、`cx` 及 `dx` 寄存器。同样，8 位寄存器来自于其 16 位的父寄存器，包括 `al`、`ah`、`bl`、`bh`、`cl`、`ch`、`dl` 及 `dh`。还有许多用于特殊寻址模式如索引寻址的特殊化的寄存器，包括 `bp`、`si` 及 `di` 寄存器。

Intel 系列的微处理器从 8088 开始，到最新的 Pentium 处理器，都是基于寄存器的。也就是说，几乎所有发生在微处理器的程序操作，都发生在一个寄存器中。唯一的例外是为浮点运算而设计的数字协处理器。协处理器是面向堆栈的，几乎所有的操作都是置于协处理器堆栈或取自于协处理器堆栈。

7.1.3 寻址模式

Intel 微处理器系列的最早成员是围绕着分片结构设计的芯片。这些最早的设计包括四个段：数据段、堆栈段、代码段及附加段。每一个段都是 64K 大小。在最近的 Pentium 微处理器设计中，允许使用线性编程模式，这样就消除了早期段设计中的大小限制，Motorola 微处理器一直使用的是线性寻址模式。

当在代码段中编写汇编语言代码时，可以使用七种基本的寻址模式，即立即寻址模式、寄存器寻址模式、直接寻址模式、寄存器间接寻址模式、相对基寻址模式、直接索引寻址模式及基索引寻址模式。通过程序语句的语法，汇编器可以识别所采用的寻址模式。例如：

```
mov    ax, 1234h    ;move 1234h into ax
```

这一程序语句是一个立即寻址的例子。此处，通过 `mov` 命令将 16 进制数 1234h 移动到了 16 位寄存器 `ax` 中。之所以认为这是一条立即寻址语句，原因是移动到 `ax` 中的数据是直接操作对象域中提供的，立即数据必须与寄存器大小匹配。

考虑如下的语句：

```
add    eax, ebx     ;add contents of ebx to eax
```

在这一语句中，使用 `add` 命令将 32 位的 `ebx` 中的内容与 32 位的 `eax` 寄存器中的内容相加。`eax` 中的原始内容被废弃，`ebx` 中的内容则保持不变，并且现在 `eax` 中存放和。由于发生了寄存器到寄存器之间的操作，所以这一编程模式被认为是寄存器寻址。在这种模式中两个寄存器必须具有相同的长度。

在直接寻址中，来自数据段的变量与寄存器一起使用。例如：

```
sub    cl, mynum     ;sub mynum from cl
```



此处的操作为从 8 位 cl 寄存器中减去变量 mynum 的值。由于涉及到要提取外部内存，所以这种直接寻址模式操作要比前面的两种寻址模式稍慢一些。同样，变量与寄存器的大小相匹配也很重要。在这一例子中，mynum 是一个字节(DB)大小的数字。

7.1.4 指针

指针给予了汇编语言程序员将一个大数据类型拆开并使用较小的寄存器保存的功能。例如，如果 myvar 定义为一个 32 位的数据类型(DD)，那么下面的代码段将使用指针加载这一完整变量到两个 16 位的寄存器中：

```
mov    ax, word ptr myvar      ;get lower 16-bit of myvar
mov    bx, word ptr myvar + 2  ;get upper 16-bit of myvar
```

指针类型必须与寄存器大小相匹配。由于字是由 2 个字节组成，为了访问 myvar 高位字，所以使用了指针值 2。

设计提示

建立指针有些复杂，但也存在技巧，指针的类型如 word、dword 和 qword 后面总是跟着 ptr 关键字，再后面是变量。指针的类型一定要与操作对象列上的寄存器类型相匹配，并且如果提供了索引值，则索引值必须与寄存器以字节表示的大小相匹配。

错误监视

指针运算是很容易引入错误的一个领域，当每次在内联代码中使用到指针运算时，一定要多检查几遍。

指针对于汇编语言程序员来说非常重要，因为它使得程序员可以对扩展了寄存器和变量长度的变量执行存取操作。注意到表 7-1 中的四字变量(DQ)没有与之直接相关的寄存器类型了否？使用指针可以将一个定义的四字变量加载到四个字长度的变量中或两双字长度的变量中。

这是汇编语言程序员的真正优势，也正因为如此才将我们带入了一个有趣的境界。大多数编译语言都受到其数据类型长度的限制，而汇编语言没有这种限制。通过指针我们可以使用 32 位、64 位或者甚至 512 位的整数。

7.1.5 协处理器

协处理器的主要工作是分担微处理器处理实数的重任(瓶颈)。实数保存在内存中，并以编码的实数在系统总线上传输。这种编码的实数被表示为非常大的整数值。请考虑：一个 32 位宽的数据总线，使用最低有效位表示单元列中的数。这就提出了一个问题，“小数点后面的数放在何处”？答案是情况并非如此。实数被转换成为一个特殊编码的整数值，该整数



值可以在数据总线上传送，在适当的时间，再将该编码的数转换回实数格式。这是一个复杂的处理过程，它降低了微处理器处理数值数据的速度——但是在协处理器中可以得到很有效的处理。

如果已经使用汇编语言编写过程序，那么协处理器的用法可能看起来有点奇怪。这是因为微处理器是面向寄存器的，而协处理器是面向堆栈的。几乎微处理器所执行的每一个操作都要涉及到处理器的内部寄存器。同样，使用协处理器所执行的几乎每一个操作也都使用到其内部堆栈。该堆栈可以容纳 8 个 80 位的数，但很少有应用程序需要使用如此深的堆栈。另外，虽然该堆栈可以容纳 80 位的数，进入协处理器的大门仍然受到一般微处理器数据类型的限制。

设计提示

协处理器允许使用命令如 **fild** 和 **fiid** 加载四字(DQ)、双字(DD)和字(DW)到其堆栈中。同样，也可以使用命令如 **fistp** 和 **fistp** 从协处理器堆栈中提取同样类型的这些数据类型。然而，显式操作如 **fadd myvar**，只有在 **myvar** 具有双字(DD)或字(DW)类型时才有效。

可以加载和保存四字数据(DQ)，或直接使用双字(DD)和字(DW)。当在协处理器内部计算时，80 位内部精度在保持精度方面发挥了其真正用途。认为协处理器只能执行实数运算是一种不正确的理解——整数值也可以使用于协处理器，只要理解了在将其置于内部堆栈时要执行 80 位实数转换即可。

在本章的稍后将看到两个协处理器应用程序，还将学习如何使用 Debugger 的 Registers 窗口观察协处理器的内部堆栈。

24x7

读者可能想到，数年以前就发现一些 Intel Pentium 微处理器已经取消了 **FDIV**、**FPREM**、**FPTAN** 及 **FPATAN** 指令。为了说明这些问题，Microsoft 在运行时库中编写了 **helper** 程序，执行精确的 **FDIV** 和 **FPREM** 计算，并提供了可以用来编写自己的 **helper** 函数的分支。

在 C++ 中，**helper** 程序被禁止使用(/QIfdiv-)。这就是说，在这些特殊的 Pentium 计算机上代码将不能正确地执行。如果 **helper** 程序打开(/QIfdiv)，则代码将测试处理器错误，并调用正确的运行时库程序。

然而，最好的解决方法是，与 Intel 联系更换带有缺陷的处理器，但是可靠的应用程序应该总是考虑到这种“错误”。

7.2 调试

内联汇编语言代码允许我们考察 Visual C++ Debugger 的独特功能——Register 窗口。虽然 Register 窗口对所有程序员都是可以使用的，但实际上这种功能最适合于调试内联汇编代



码的情况。使用这一窗口我们可以观察微处理器的寄存器和协处理器的堆栈情况。

在下面的几节中，我们将检查从简单应用程序到更可靠的协处理器实例中的内联汇编代码。这些代码中起初大多都包含逻辑错误，使得应用程序不能象希望的那样运行。我们将使用 Debugger 帮助捕获这些逻辑错误，并最终修复这些应用程序。只有包含“contains no errors” (不包含错误)注释的应用程序可以正常运行。

另外，如果读者需要提高其汇编语言编程技能，那么笔者建议参考其他专门的汇编语言图书。

7.2.1 减法运算

本节中所演示的实例不包含错误。我们将使用这一实例作为一个台阶，以成功编写内联汇编代码并研究各种调试特性。

下面是 Sub.cpp 程序的完整清单。

```
// Sub.cpp
// Assembly Language Program
// illustrates debugger use
// contains no errors
#include <iostream.h>
void main(void)
{
    unsigned int num_1=0xABCD8765;
    unsigned int num_2=0x12345678;
    unsigned int result;
    _asm {
        ;code to perform actual subtraction
        mov     eax,num_1    ;move num_1 into eax register
        sub     eax,num_2    ;subtract num_2 from eax
        mov     result,eax   ;store eax in result
    }
    cout << hex << result << "\n";
}
```

在这一小应用程序中实际上只包含了三行汇编语言代码，但其中包含了关于编写内联代码的许多信息。

首先，汇编语言代码包含在 `_asm` 关键字下的大括号({})之间：

```
_asm {
    .
    .
    .
}
```



}

这一部分中所包含的代码必须以汇编语言格式书写，并且必须与开发汇编语言代码所使用的四个域匹配。这四个域是名称(标记)域、操作(记忆符)域、操作对象域及注释域。在这一例子中，只使用了操作、操作对象和注释域。注意，注释总是以分号开始。

这一应用程序使用 32 位的 `eax` 寄存器执行两个十六进制数的减法运算。虽然通常我们使用的是十进制数，但十六进制是汇编语言程序员所选择的进制系统。

关于参加减法运算的数值的数据声明在内联汇编语言代码之外给出，所以，声明它们与其他 C 或 C++ 整数一样。

```
unsigned int num_1=0xABCD8765;
unsigned int num_2=0x12345678;
unsigned int result;
```

有一个技巧就是使 C++ 数据类型正确地与汇编语言的数据类型匹配。由于 `eax` 寄存器容纳 32 位的整数，所以 C++ 的 `int` 数据类型与之正好适合。

错误监视

在这一例子中，三个数据值均声明为无符号(unsigned)，这是因为汇编符号扩展数的原因。例如，从 0 到 7FFFFFFF 范围内的十六进制值，在执行有符号运算时被看作是正数，而在 80000000 到 FFFFFFFF 范围内的值则被认为是负数。如果声明数值为无符号的，则在 0 到 FFFFFFFF 范围内的数均被认为是正数。

通过将数值看作为无符号的，我们即可期待无符号十六进制数的结果为 999930ED。

7.2.1.1 好的代码

前一节中给出的例子不包含错误。我们将使用这一例子检查 Visual C++ Debugger 究竟可以为我们做些什么。打开 Debug 选项(缺省)编译该程序，并进入 Debugger，将看到一个与图 7-1 类似的屏幕。

在该图中可以看出，由箭头符号所指的执行代码行出现在屏幕的最左边。初始的 Registers 窗口显示了微处理器寄存器和协处理器堆栈。在这一例子中，我们所感兴趣的是在 `eax` 寄存器中所发生的情况，因为在此处要执行减法运算。

单步执行该应用程序，使用 Step Into 按钮(F11)，直到执行箭头指向图 7-2 所示的位置。

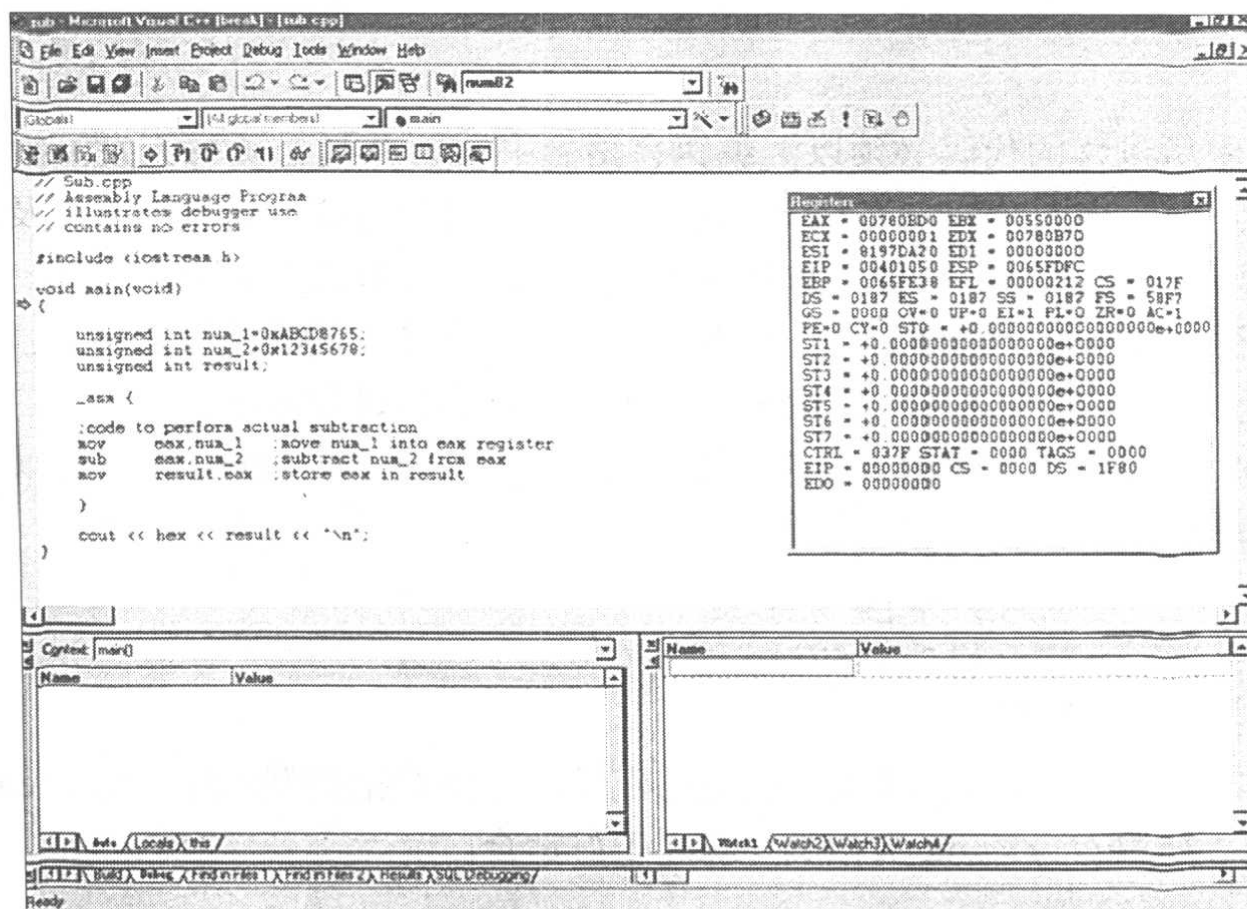


图 7-1 显示 Sub.cpp 程序的初始 Debugger 屏幕

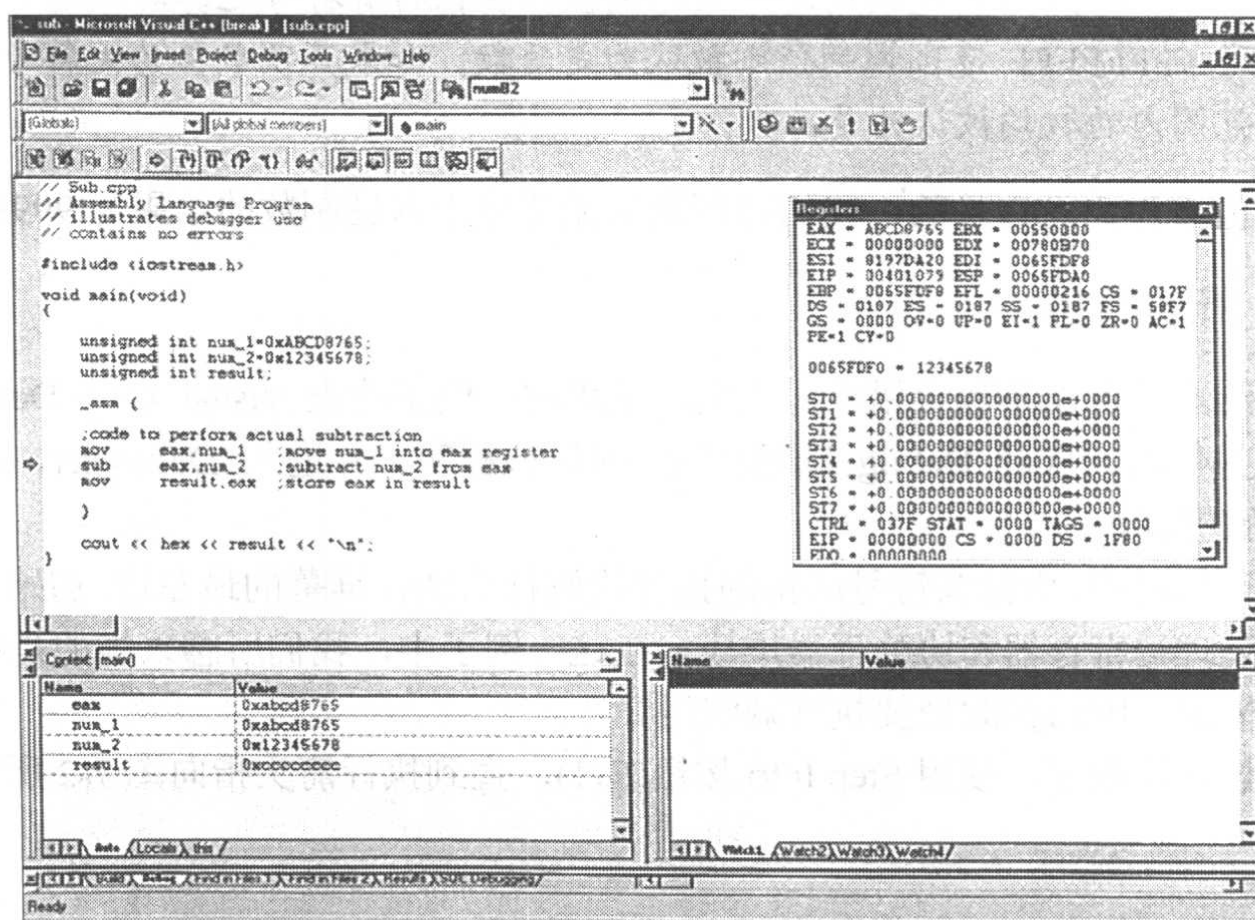


图 7-2 第一个十六进制数加载到 eax 寄存器



注意，在图 7-2 中，`eax` 寄存器中包含的值为 `ABCD8765`。在 `Variables` 窗口(在左下部)和 `Registers` 窗口中都可以看到这一值。Watch 窗口(在图 7-2 的右下部)中没有包含任何内容。如果这些窗口中有不可见的，可使用 `View|Debug Windows` 菜单打开所需的窗口。

前进到下一代码行，使用 `Step Into` 按钮(F11)，执行实际的减法操作。此时的屏幕显示将类似于图 7-3 所示。

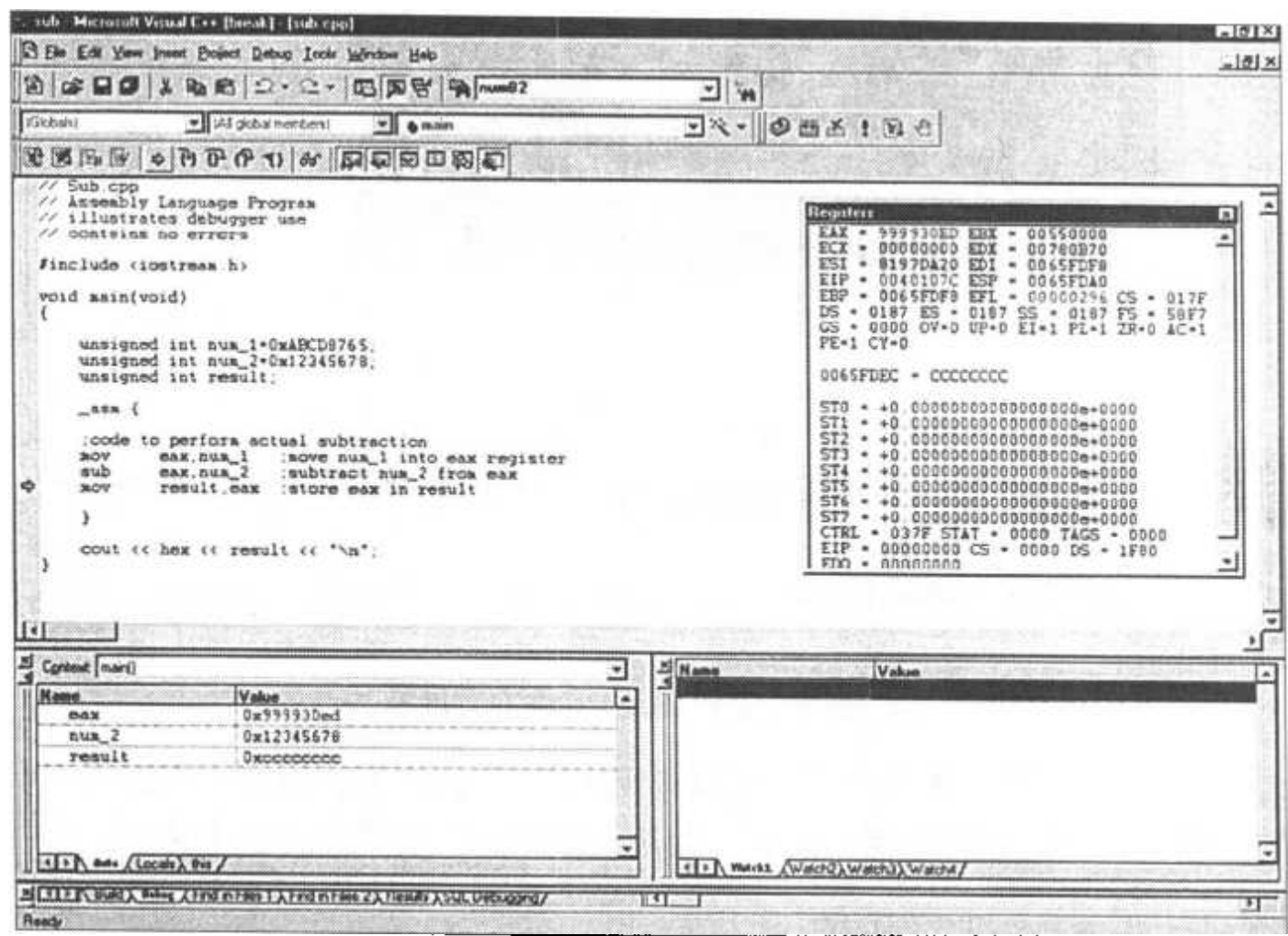


图 7-3 `eax` 寄存器中包含了减法运算的结果

减法运算成功，现在正确的结果出现在 `eax` 寄存器中。同样，也可以在 `Registers` 和 `Variables` 窗口中同时看到该寄存器。通过单击 `Variables` 窗口中的任何一个变量，Debugger 将给出值的观察方式选项：十六进制或十进制。

由于该应用程序没有使用到协处理器，所以从 `ST(0)`到 `ST(8)`的协处理器堆栈项中都没有包含有意义的值。

由于变量 `result` 是 C++数据类型，其中所保存的值可以打印到屏幕上，所以只要使用 `cout` 流，将结果以十六进制打印即可。如果执行该应用程序，则将在与图 7-4 类似的窗口内看到这一结果。

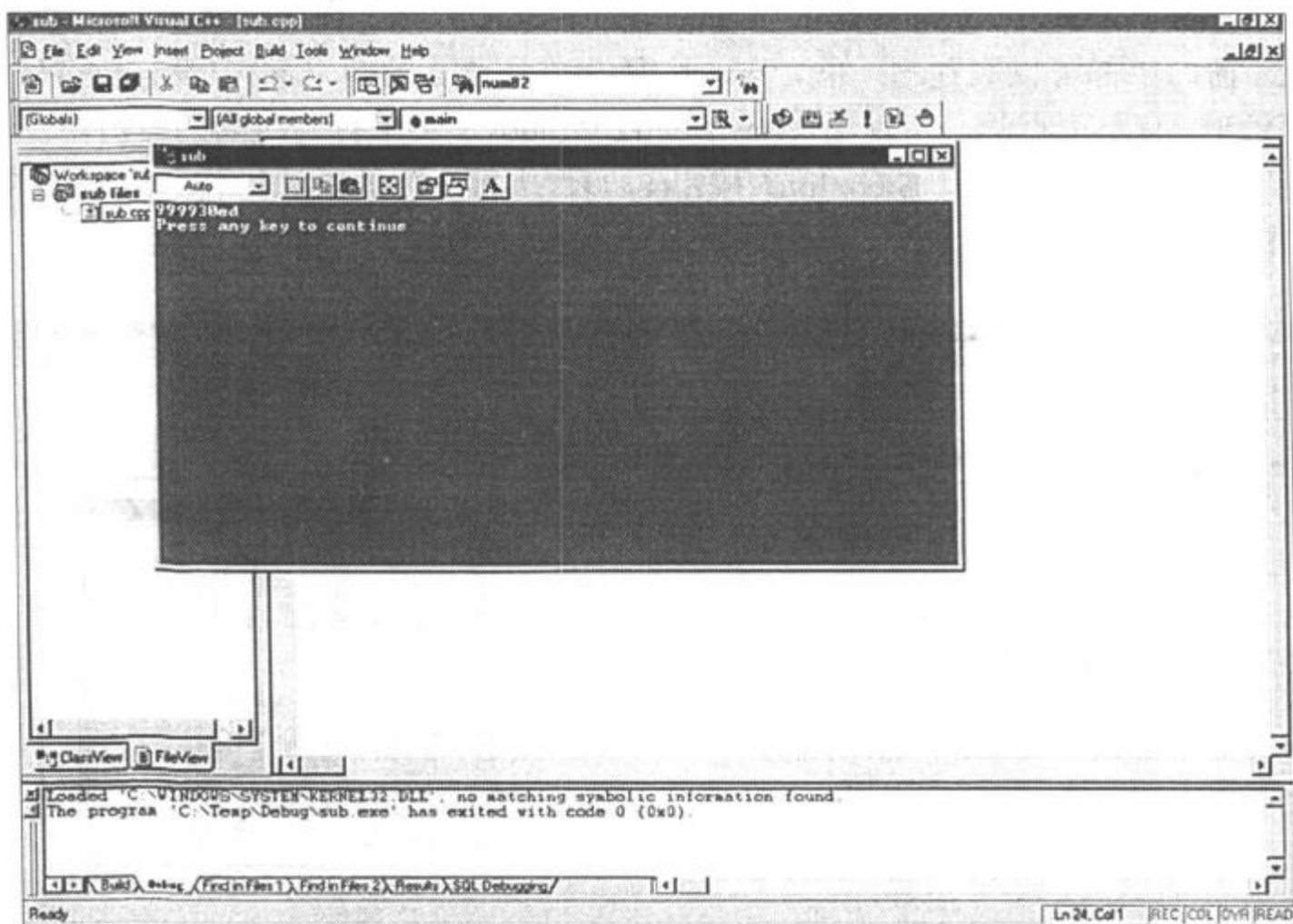


图 7-4 结果可以打印到一个命令行屏幕上

可以看出，从 C++ 代码移动到汇编再移动回 C++ 非常容易，这使得汇编语言代码成为了注重时间的应用程序的理想选择。然而，使用汇编语言还有其他方面的原因，在下一个例子中将看到。

7.2.2 使用 256 位整数

汇编语言允许使用多种数据长度，即使数据长度超过所有已定义的数据类型。在这一例子中，我们演示了这种多重精度算术运算的实现方法，该程序对两个 256 位的数执行加法运算。由于使用公共密钥加密学加密程序需要保存和操作大素数的缘故，所以这种长度的数或更大长度的数变得越来越重要了。

此处所采用的技巧是，将大数分解为已定义数据类型可以存放的较小的段。例如，如下所示的 256 位的数，可以分解为四个适合于无符号 `__int64` 整数数据类型的段。

```
//1234567891234567ABCDABCDABCDABCDFFFFEEEEEDDDDDCCCCDDDDCCCCBBBBBAAAA
unsigned __int64 numA1 = 0xDDDDCCCCBBBBBAAAA;
unsigned __int64 numA2 = 0xFFFFEEEEEDDDDDCCCC;
unsigned __int64 numA3 = 0xABCDABCDABCDABCD;
unsigned __int64 numA4 = 0x1234567891234567;
```



然后通过使用指针，将各个段相加，并将其部分和返回到四个无符号 `__int64` 变量中。下一节中，我们将看到这一程序的一个优化编码，然后使用 Debugger 捕获程序错误。

7.2.2.1 带有问题的代码

在分析该程序如何处理之前，让我们首先计算这一结果。如果我们不知道答案是什么样子，那么在 Debugger 中观察时我们也永远不会知道其是否正确。

```
1234567891234567ABCDABCDABCDABCDFFFFEEEEEDDDDDCCCCDDDDCCCCBBBBBAAAA
22222222222222221111111111111111A1B2C3D4E5F6E5D48888999911112222
-----
3456789AB3456789BCDEBCDEBCDEBCDFA1B2B2C3C3D4B2A166666665CCCCCCCC
```

这确实是一个很大的数。我们的计算器一次可处理多少位？

基本的编程处理方法是，通过分别相加四个数据段的方法，对这两个数相加。

```
numA4 numA3 numA2 numA1
numB4 numB3 numB2 numB1
-----
ANS4  ANS3  ANS2  ANS1
```

所以，对于 `numA1` 和 `numB1` 相加的第一个加法，其代码形式如下：

```
;start with numA1 and numB1
mov     eax,dword ptr numA1           ;get first 8 digits
add     eax,dword ptr numB1           ;add first 8 digits
mov     dword ptr ANS1,eax            ;save first 8 digits
mov     eax,dword ptr numA1 + 4       ;get next 8 digits
add     eax,dword ptr numB1 + 4       ;add next 8 digits + carry
mov     dword ptr ANS1 + 4,eax        ;save next 8 digits
```

数 `numA1` 和 `numB1` 是 64 位的整数，必须使用指针在 32 位 `eax` 寄存器中完成加法运算。程序代码的前两行使用一个 `dword` 指针求 `numA1` 和 `numB1` 的前 8 位数字(32 位)的和，这一 32 位的结果保存在 `ANS1` 的最低位。`numA1` 和 `numB1` 的最高 32 位使用一个偏移量为 4(32 位)的 `dword` 指针相加，这一结果保存在 `ANS1` 的最高位。我们现在完成了前 64 位的加法运算。

为求 `numA2` 和 `numB2`、`numA3` 和 `numB3`、`numA4` 和 `numB4` 的和重复同样的过程。当所有加法运算完成后，256 位的和将出现在 `ANS4`、`ANS3`、`ANS2` 和 `ANS1` 中。

下面是完整的程序：

```
// Contains Logical Errors
// Assembly Language Program
// illustrates the power of assembly language in
```



```
// correctly manipulating 256-bit numbers
// with multiple precision arithmetic.
void main(void)
{
//break up and store the first 256-bit integer number
//1234567891234567ABCDABCDABCDABCDFFFFFFFFEEEEDDDDCCCCDDDDCCCCBBBBBAAAA
    unsigned __int64 numA1 = 0xDDDDCCCCBBBBBAAAA;
    unsigned __int64 numA2 = 0xFFFFFFFFEEEEDDDDCCCC;
    unsigned __int64 numA3 = 0xABCDABCDABCDABCD;
    unsigned __int64 numA4 = 0x1234567891234567;
//break up and store the second 256-bit integer number
//22222222222222221111111111111111A1B2C3D4E5F6E5D48888999911112222
    unsigned __int64 numB1 = 0x8888999911112222;
    unsigned __int64 numB2 = 0xA1B2C3D4E5F6E5D4;
    unsigned __int64 numB3 = 0x1111111111111111;
    unsigned __int64 numB4 = 0x2222222222222222;
//prepare storage locations to hold the 256-bit result
    unsigned __int64 ANS1 = 0;
    unsigned __int64 ANS2 = 0;
    unsigned __int64 ANS3 = 0;
    unsigned __int64 ANS4 = 0;
    _asm {
        ;start with numA1 and numB1
        mov eax,dword ptr numA1      ;get first 8 digits
        add eax,dword ptr numB1      ;add first 8 digits
        mov dword ptr ANS1,eax       ;save first 8 digits
        mov eax,dword ptr numA1 + 4  ;get next 8 digits
        add eax,dword ptr numB1 + 4  ;add next 8 digits + carry
        mov dword ptr ANS1 + 4,eax    ;save next 8 digits
        ;work with numA2 and numB2 + carry information
        mov eax,dword ptr numA2      ;get next 8 digits
        add eax,dword ptr numB2      ;add next 8 digits + carry
        mov dword ptr ANS2,eax       ;save next 8 digits
        mov eax,dword ptr numA2 + 4  ;get next 8 digits
        add eax,dword ptr numB2 + 4  ;add next 8 digits + carry
        mov dword ptr ANS2 + 4,eax    ;save next 8 digits
        ;work with numA3 and numB3 + carry information
        mov eax,dword ptr numA3      ;get next 8 digits
        add eax,dword ptr numB3      ;add next 8 digits + carry
        mov dword ptr ANS3,eax       ;save next 8 digits
        mov eax,dword ptr numA3 + 4  ;get next 8 digits
        add eax,dword ptr numB3 + 4  ;add next 8 digits + carry
        mov dword ptr ANS3 + 4,eax    ;save next 8 digits
    }
```

1

[illegible]

初看起来观察窗口中完整的答案没有什么问题，该答案为：

⤴ ⤵ ⤶



三个脱字符(^)所在的位置所指定的数字与我们的计算结果不同。现在,在一般情况下,可能是我们自己计算错误而程序的返回结果正确。但此处却恰恰相反。

现在,我们返回到 Debugger 并开始单步执行代码的每一行,检查其结果。

图 7-6 给出了计算第一个 64 位相加后微处理器寄存器的情况。

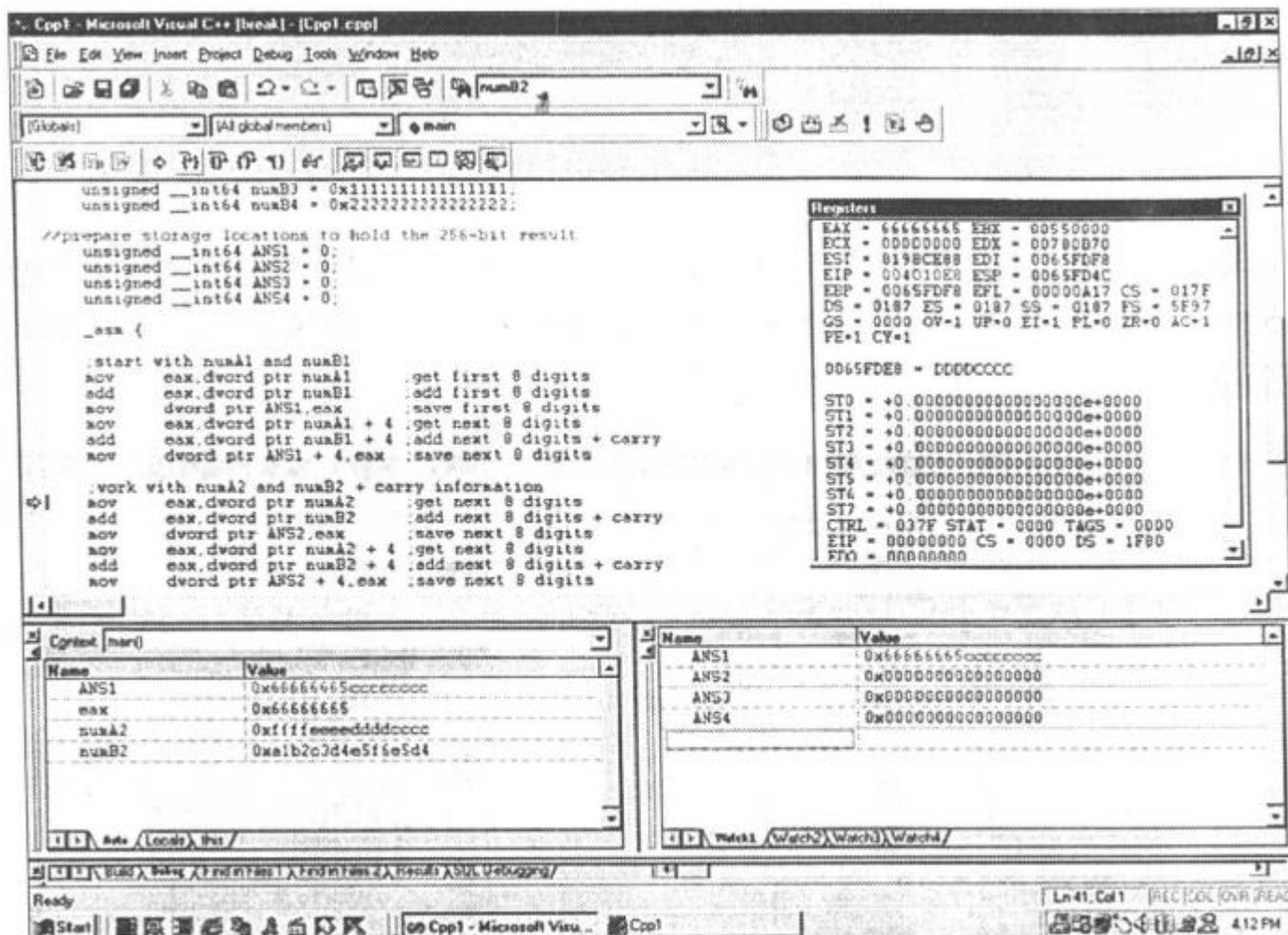


图 7-6 第一个 64 位加法正确

变量 ANS1 清楚地显示一个十六进制值 66666665cccccccc。这正是我们的计算结果,所以此处不存在问题。

现在,执行第二个 64 位相加,调试器屏幕如图 7-7 所示。

问题出在这一部分,我们应该得到的十六进制数为 A1B2B2C3C3D4B2A1,而现在我们得到的是 A1B2B2C2C3D4B2A0。

是否注意到在每一个 32 位数的最低位都发生了错误?这一部分的代码有什么问题否?问题是这样的,当从加法的一部分移动到另一部分时,这些数字位应该从前一列的和接收一个进位。让我们查看进位标志是否在工作。在 Registers 窗口中可以找到进位标志(CY)。表 7-2 列出了微处理器的所有标志。

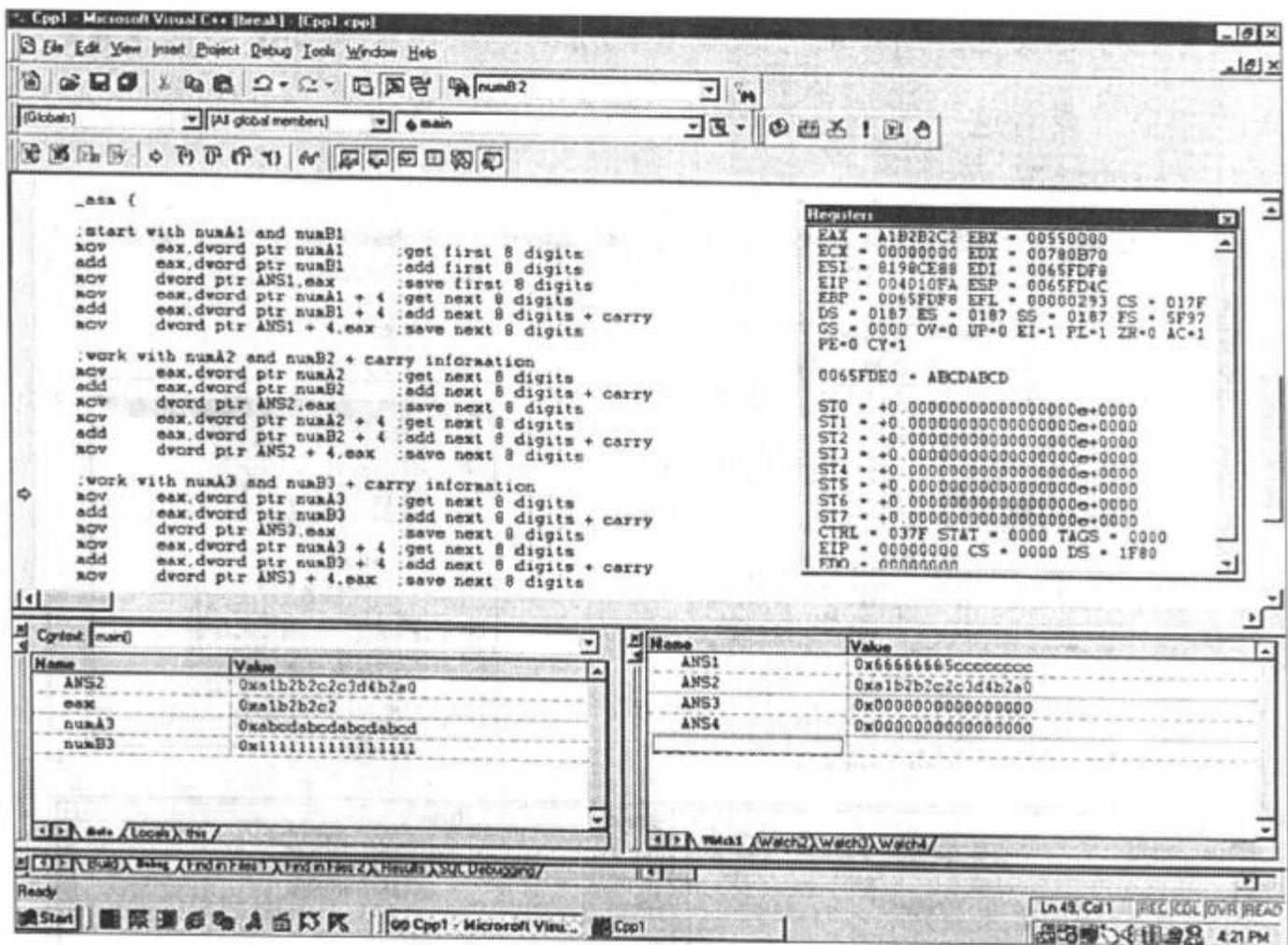


图 7-7 第二个 64 位加法不正确

表 7-2 微处理器标志

助记符	标志	设置时为
OV	溢出	1
UP	方向	1
EI	中断	1
PL	符号	1
ZR	零	1
AC	辅助进位	1
PE	奇偶校验	1
CY	进位	1

再回到图 7-6 并注意在 64 位加法之后，CY 标志设置。这意味着应该向第二个 64 加法的第一个数字列加 1，但实际并未这样做。我们的总计在最低位上小 1，也就是说，程序没有考虑进位信息。为什么？

在汇编语言中，有两个加法命令：**add** 和 **adc**。

设计提示

add 命令使用于没有进位信息的情况。这将是 256 位数中最低位数字的情况。在这之后，



进位标志每次都有可能设置。所以对于所有其他情况,应该使用 **adc** 加法命令。

正确修改命令,查看这一纠正是否适用于整个程序。图 7-8 给出了第三组 64 位数相加后微处理器寄存器的情况。

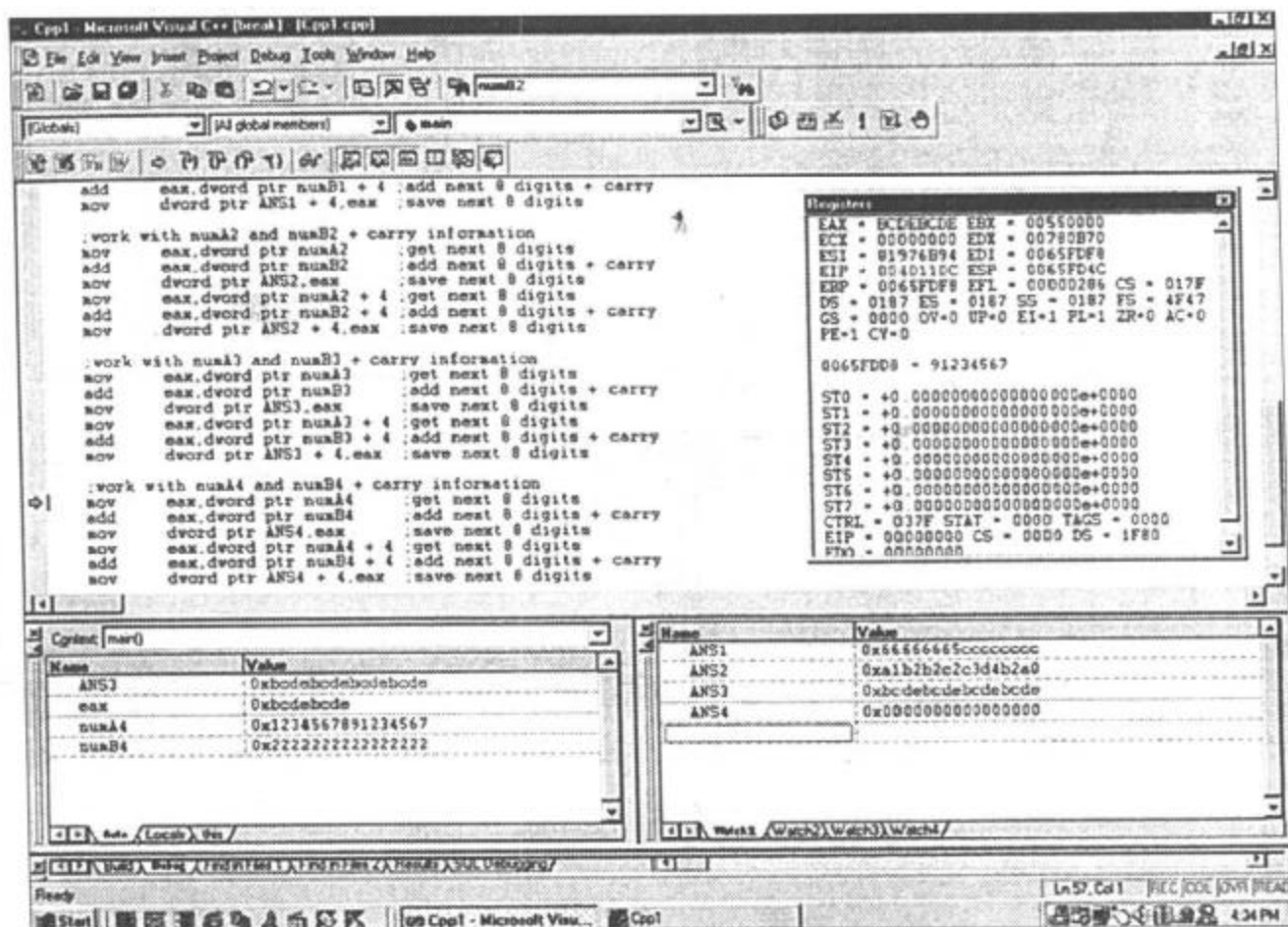


图 7-8 该 64 位的相加结果仍不正确

当然,ANS3 的最低位数字应该是十六进制数“F”,但此处为“E”。我们现在知道这是进位问题造成的,因为在该加法运算之前进位标志(CY)是设置的,见图 7-7。

最后,256 位数字的最高 64 位的加法是正确的,如图 7-9 所示。

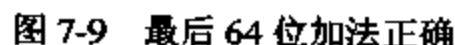
这一加法正确的原因是前一 64 位加法没有进位信息,可以从图 7-8 中查看进位标志(CY)即可。确实如此,进位标志没有设置。

我们需要如何纠正这一程序?下一节中我们将看到该程序的修改情况。

7.2.2.2 好的代码

正如在前一节中所介绍的那样,该程序的问题在于没有能在多重精度运算的各个阶段都引用进位信息。使用 Debugger 可以很容易地看出这一问题,并同时注意相加的结果和进位标志(CY)。

只有第一个加法使用 **add** 命令,其他的加法必须使用 **adc** 命令,因为所有后面的加法都有可能使用到进位信息。



```
// BigInt.cpp
// Assembly Language Program
// illustrates the power of assembly language in
// correctly manipulating 256-bit numbers
// with multiple precision arithmetic.
// contains no errors
void main(void)
{
//break up and store the first 256-bit integer number
//1234567891234567ABCDABCDABCDABCDFFFFEEEEEDDDDDCCCCDDDDCCCCBBBBBAAAA
    unsigned __int64 numA1 = 0xDDDDCCCCBBBBBAAA;
    unsigned __int64 numA2 = 0xFFFFEEEEEDDDCCCC;
    unsigned __int64 numA3 = 0xABCDABCDABCDABCD;
    unsigned __int64 numA4 = 0x1234567891234567;
//break up and store the second 256-bit integer number
//2222222222222222221111111111111111A1B2C3D4E5F6E5D48888999911112222
    unsigned __int64 numB1 = 0x8888999911112222;
    unsigned __int64 numB2 = 0xA1B2C3D4E5F6E5D4;
    unsigned __int64 numB3 = 0x1111111111111111;
    unsigned __int64 numB4 = 0x2222222222222222;
```



```
//prepare storage locations to hold the 256-bit result
unsigned __int64 ANS1 = 0;
unsigned __int64 ANS2 = 0;
unsigned __int64 ANS3 = 0;
unsigned __int64 ANS4 = 0;
_asm {
;start with numA1 and numB1
mov eax,dword ptr numA1      ;get first 8 digits
add eax,dword ptr numB1      ;add first 8 digits
mov dword ptr ANS1,eax       ;save first 8 digits
mov eax,dword ptr numA1 + 4  ;get next 8 digits
adc eax,dword ptr numB1 + 4  ;add next 8 digits + carry
mov dword ptr ANS1 + 4,eax    ;save next 8 digits
;work with numA2 and numB2 + carry information
mov eax,dword ptr numA2      ;get next 8 digits
adc eax,dword ptr numB2      ;add next 8 digits + carry
mov dword ptr ANS2,eax       ;save next 8 digits
mov eax,dword ptr numA2 + 4  ;get next 8 digits
adc eax,dword ptr numB2 + 4  ;add next 8 digits + carry
mov dword ptr ANS2 + 4,eax    ;save next 8 digits
;work with numA3 and numB3 + carry information
mov eax,dword ptr numA3      ;get next 8 digits
adc eax,dword ptr numB3      ;add next 8 digits + carry
mov dword ptr ANS3,eax       ;save next 8 digits
mov eax,dword ptr numA3 + 4  ;get next 8 digits
adc eax,dword ptr numB3 + 4  ;add next 8 digits + carry
mov dword ptr ANS3 + 4,eax    ;save next 8 digits
;work with numA4 and numB4 + carry information
mov eax,dword ptr numA4      ;get next 8 digits
adc eax,dword ptr numB4      ;add next 8 digits + carry
mov dword ptr ANS4,eax       ;save next 8 digits
mov eax,dword ptr numA4 + 4  ;get next 8 digits
adc eax,dword ptr numB4 + 4  ;add next 8 digits + carry
mov dword ptr ANS4 + 4,eax    ;save next 8 digits

}
}
```

使用 Debugger 单步运行所有的加法运算，观察 Watch 窗口的结果，屏幕显示应类似于图 7-10 所示。

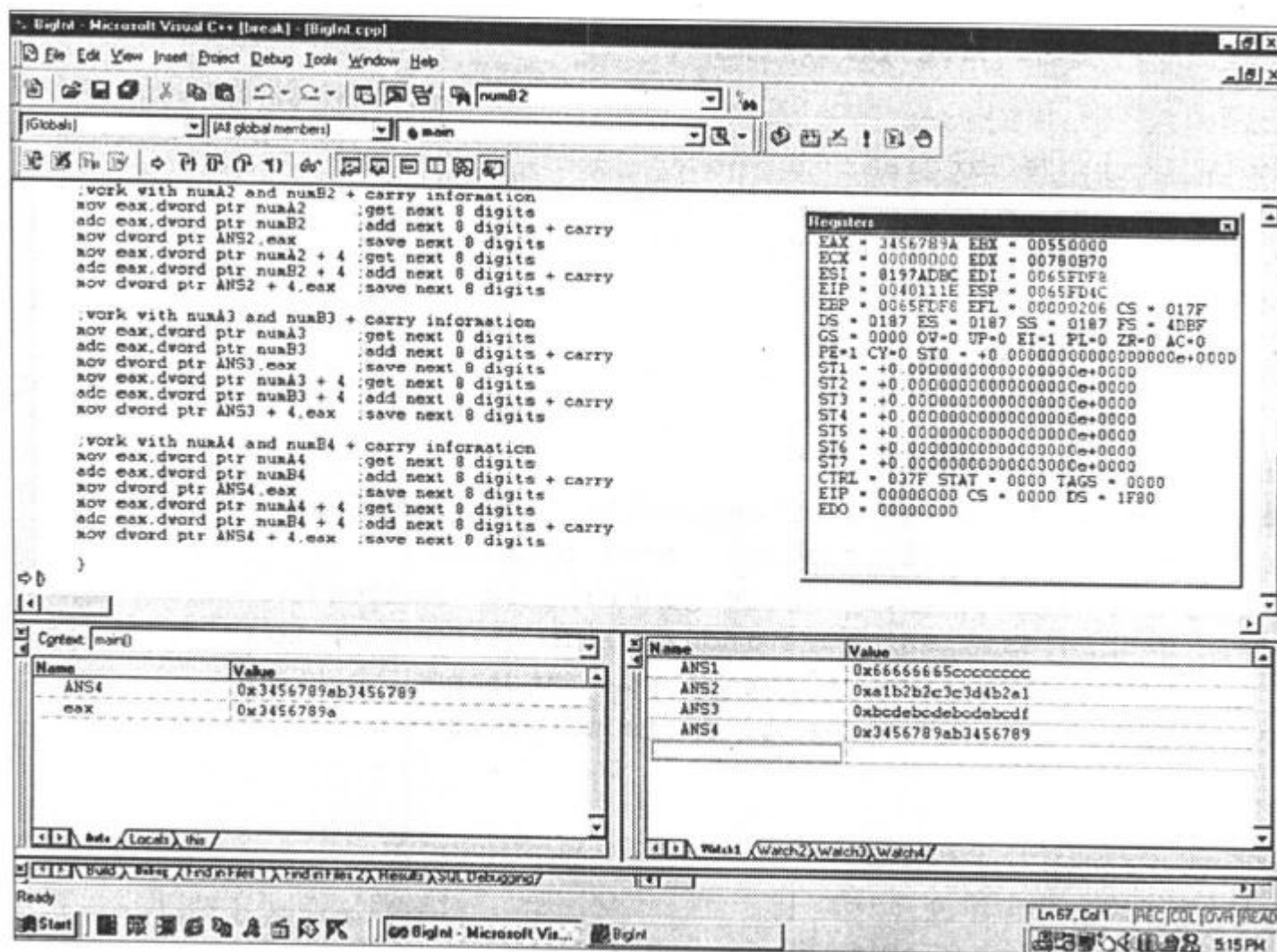


图 7-10 正确的 256 位和出现在 Watch 窗口中

在执行多重精度运算时，指针只是一个汇编语言程序员预先考虑问题的一小部分。在试图跟踪程序错误时，命令之间的微弱差别，如 **add** 和 **adc** 或 **sub** 和 **sbb**，可以造成严重错误。

7.2.3 程序循环

所有的编程语言都提供了一些循环控制手段。在 C++ 中有我们熟悉的 **for**、**while** 和 **do-while** 循环。汇编语言也有多种方法实现循环，但最普遍的应该是使用 **loop** 命令。当需要完成重复性任务时，在任何语言中循环都非常有用。有各种各样的这种重复性任务，从使用指定的数据填写一个表，到中断一个程序刷新整个屏幕。

下一节中所描述的例子声明了一个 C++ 数组，其中可以保存 100 个整型值。汇编语言代码将使用索引寻址模式，以顺序增加的偶数值填写这一数组。当填写该数组后，将使用 **cout** 流在屏幕上打印数组的内容。

7.2.3.1 存在问题的代码

这个应用程序的编程代码比较简单——没有多重精度运算。然而，在这一简单的程序



中存在一些微妙的错误。首先查看原始实现。

```
// Contains Logical Errors
// Assembly Language Program
// illustrates the use of a loop in assembly language
// to fill a table with sequentially higher numbers.
#include <iostream.h>
void main(void)
{
    int etable[100];
    _asm {
        ;code to generate a table of sequential even numbers
        mov     esi,00h      ;initialize index register
        mov     ecx,100      ;prepare to generate 100 numbers
more: mov     etable[esi],esi ;use index value as number, too
        add     esi,04h      ;now point to next storage location
        loop    more        ;repeat if cx is not zero
    }
    // print values in table
    for (int i=0; i<100 ; i++) {
        cout << hex << etable[i] << "\n";
    }
}
```

在这一例子中，使用索引寻址访问数组(在汇编语言中经常称之为表)中的各个位置。所以，第一步设置索引寄存器为起始位置，循环控制基于 `ecx` 寄存器中的值执行。当执行循环时 `ecx` 中的值将自动在每次循环经过时减少。当 `ecx` 中的值为 0 时，控制退出循环，程序中的汇编语言部分结束。为了使用汇编语言从一个整数增加到另一个整数，在每一次循环期间将一个 `8h` 的值添加到 `esi` 寄存器中。经过明显而有效的移动，放置在表中的数据值就是当前包含在 `esi` 寄存器中的值。我们所看到的将是十六进制的表项如 0、8、10、18、20 等。

编译该程序，并进入 Debugger 检查其结果。将变量 `etable` 放入 Watch 窗口，如图 7-11 所示。

同样，注意已经有若干个代码行执行完毕。事实上，只执行了循环的第一轮。检查图 7-11，可以发现在 Watch 窗口中的 `etable` 左边有一个小加号(+)。单击这一符号，可展开 `etable` 的内容，如图 7-12 所示。

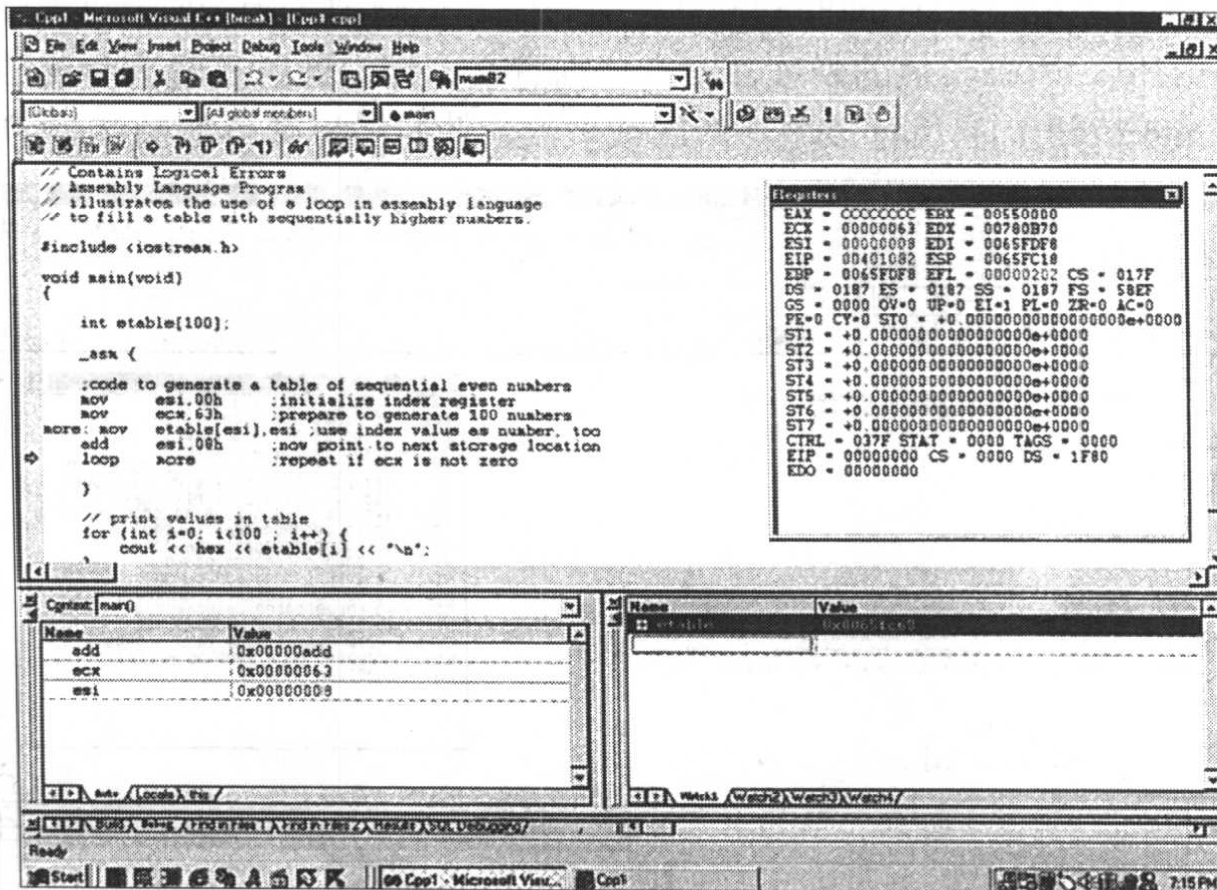


图 7-11 将变量 etable 放入 Watch 窗口

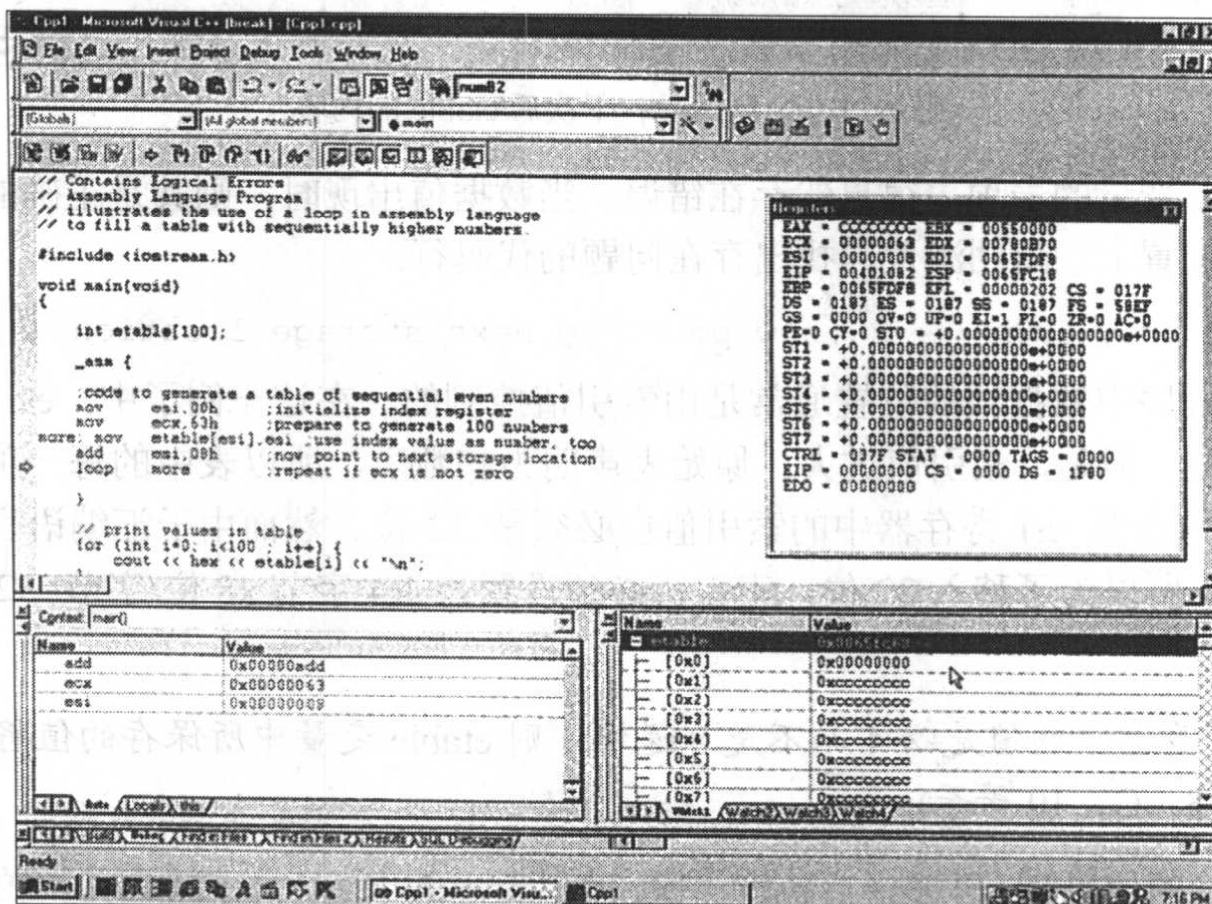


图 7-12 在 Watch 窗口中可以看到 etable 的内容



第一项 `etable[0x0]` 的值为 `0x00000000`。这一值是正确的，因为它代表的是第一次循环时 `esi` 寄存器中的值。所有其他的表项都没有任何意义，因为此时还没有将值放在此处。

使用 **Setp Into** 按钮(F11)执行该循环两次或多次，此时的屏幕将如图 7-13 所示。



图 7-13 循环执行若干次后 `etable` 的内容

检查 `etable` 变量的表项，很显然存在错误。当数据值出现时，则其是正确的，但都出现在每隔一个的位置上。返回源代码找出存在问题的代码行。

```
add     esi, 08h           ;now point to next storage location
```

汇编语言程序中数据项的位置通常是由索引值控制的。在这一例子中，`esi` 寄存器用于索引所有的表项。但是，索引值过大。原始表声明为整数表，所以表中的每一项均将是一个 32 位的整数。添加到 `esi` 寄存器中的索引值也必须是 32 位。然而由于汇编语言中的索引值是以字节计算，所以为了移入 32 位，索引值必须设置为 4(4 字节 x 8 位/字节=32 位)。

设计提示

如果 `esi` 寄存器中的值是以 4 而不是 8 递增，则 `etable` 变量中所保存的值将以 4 的倍数递增(即 0、4、8、C、10 等等)。

使用了正确的值以后，再编译后执行一次，可以看出表中输入的值都是正确的，如图 7-14 所示。

将循环执行到底，检查图 7-15，可以看出另一个问题。

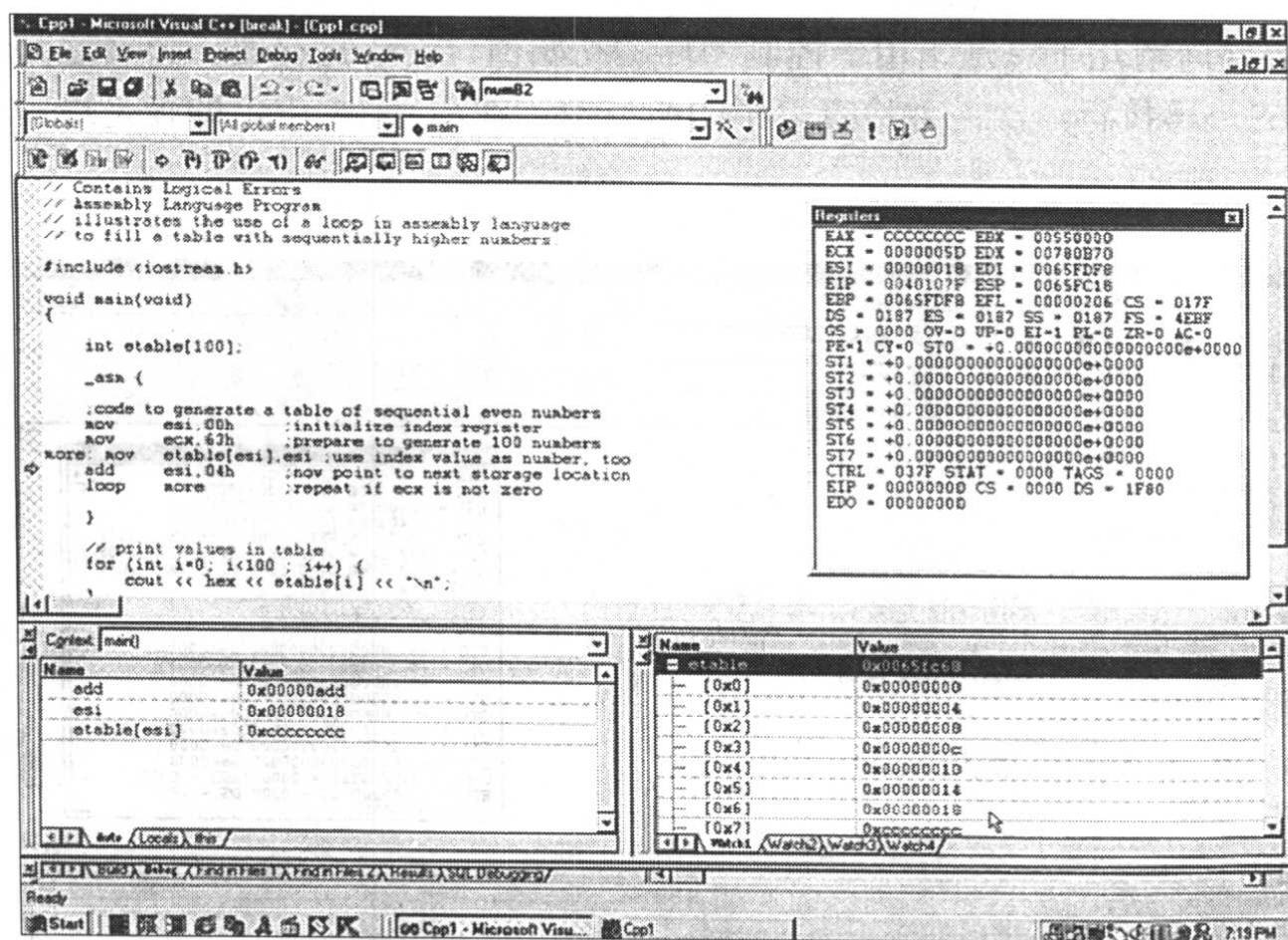


图 7-14 现在 etable 的值正确地递增和保存

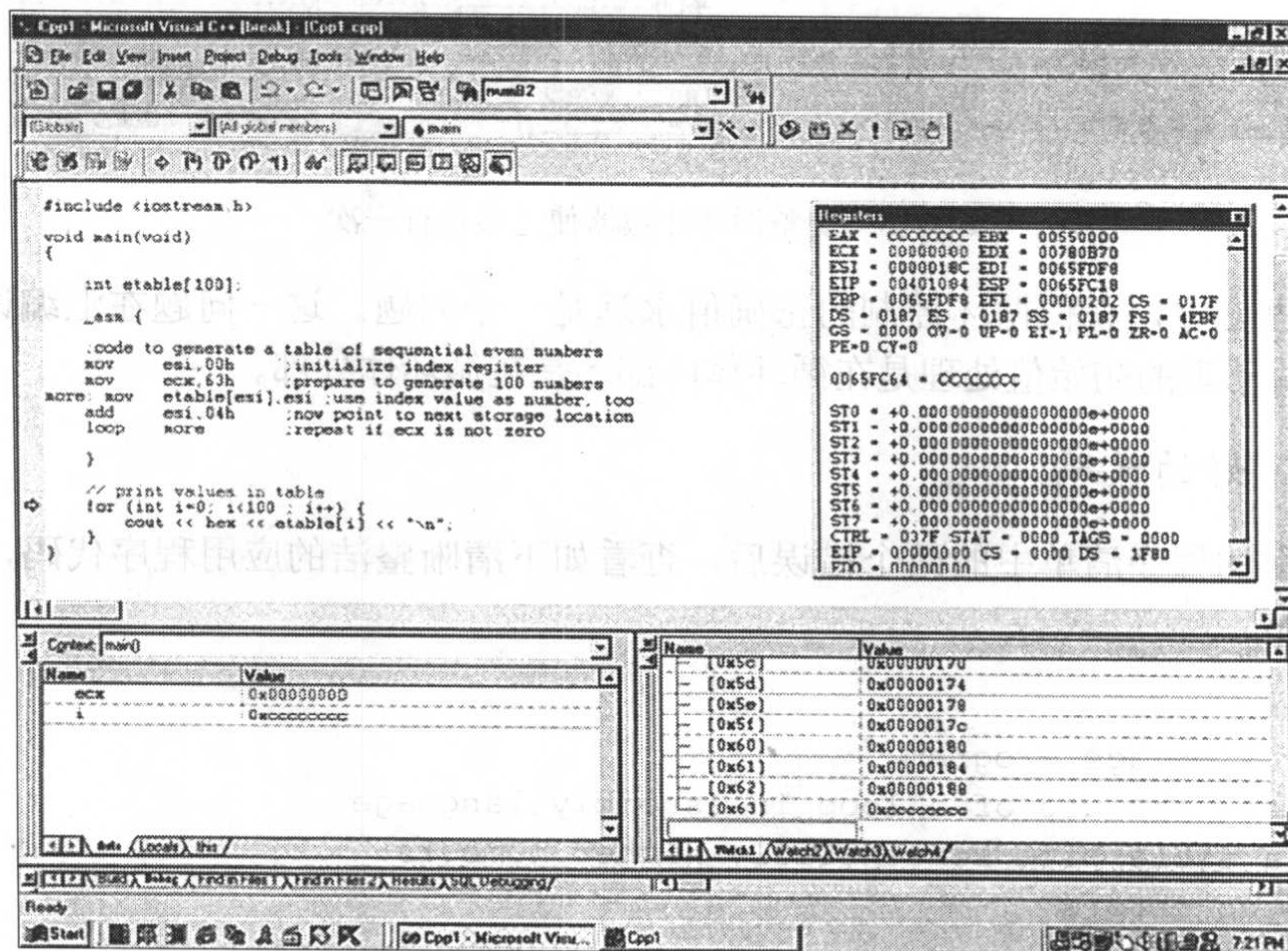


图 7-15 表中最后一个值[0x63]没有输入



循环的边界问题几乎总是指用于控制循环的初始值，ecx 寄存器应该设置为十六进制的 64，而不是 63。再执行一次，并检查表中的所有值。在图 7-16 中，即使是最后一个值也是正确的。



图 7-16 调整循环计数器使之多执行一次

在任何语言中，获得循环控制的正确值永远是一个问题。这一问题在汇编语言中尤为严重，这取决于表的初始值处理是在循环体内部还是在循环体外部。

7.2.3.2 好的代码

纠正前面的程序清单中的两个错误后，查看如下清晰整洁的应用程序代码，并将其命名为 Loop.cpp。

```
// Loop.cpp
// Assembly Language Program
// illustrates the use of a loop in assembly language
// to fill a table with sequentially higher numbers.
// contains no errors
#include <iostream.h>
void main(void)
{
```




```
int etable[100];
_asm {
;code to generate a table of sequential even numbers
mov     esi,00h           ;initialize index register
mov     ecx,64h           ;prepare to generate 100 numbers
more:   mov     etable[esi],esi ;use index value as number, too
        add     esi,04h     ;now point to next storage location
        loop    more       ;repeat if cx is not zero
}
// print values in table
for (int i=0; i<100 ; i++) {
    cout << hex << etable[i] << "\n";
}
```

最后，检查该应用程序的性能，执行该应用程序，并检查使用 cout 流打印在兼容框中的数据，如图 7-17 所示。

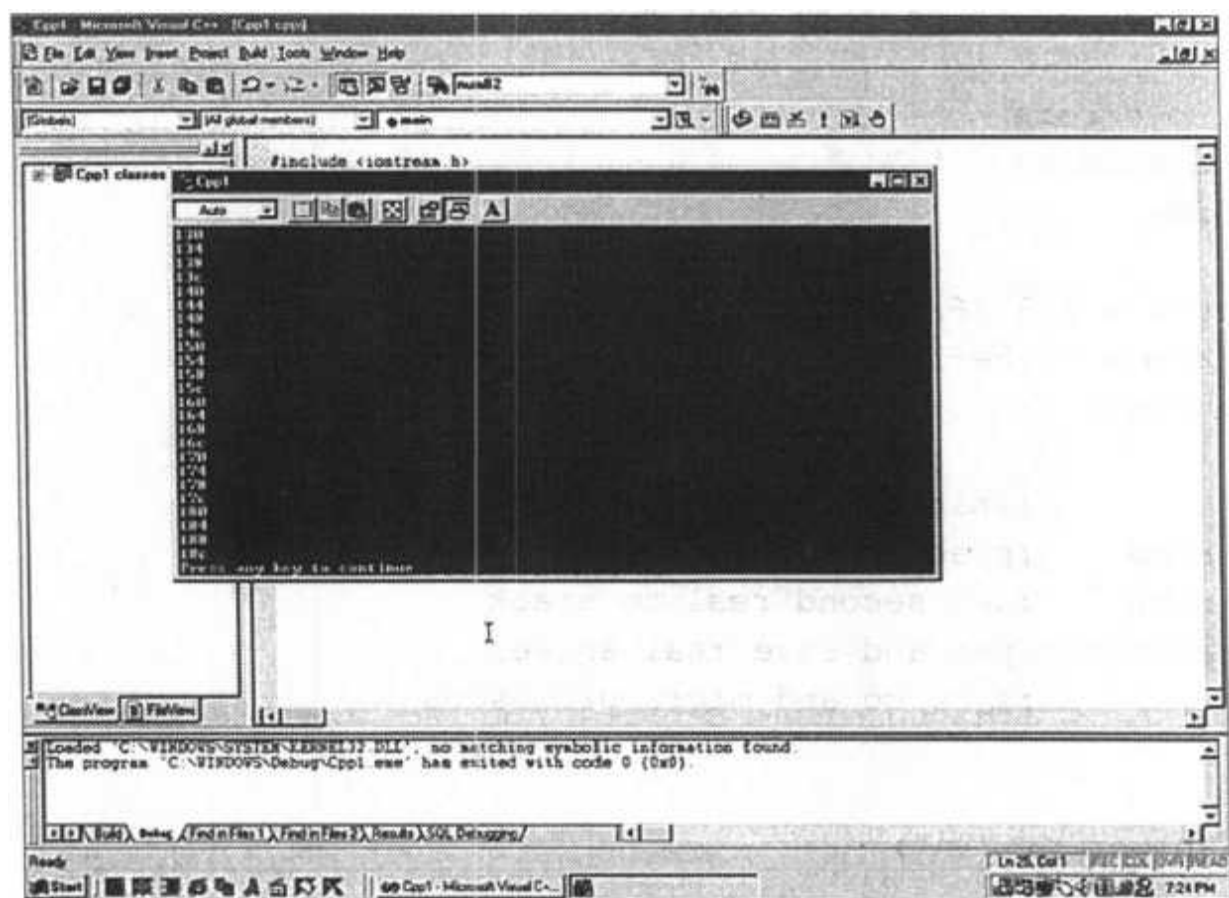


图 7-17 打印到屏幕上的 Loop.cpp 应用程序的输出

切记，虽然这些值是按照要求以十六进制数打印，但也可以使用普通的十进制形式将其打印到屏幕上。

到现在为止，所有的汇编语言程序都处理的是整形数。下一节中，我们将学习如何使



用协处理器执行实数运算。还将学习如何结合 Debugger 的 Registers 窗口，观察协处理器堆栈的添加和删除。

7.2.4 使用协处理器求和实数

演示协处理器的第一个程序中不包含任何错误。我们将使用这一程序介绍一些协处理器编程的基础，并介绍如何使用调试器检查变量和协处理器堆栈项。

该程序为 RealAdd.cpp，这是一个将两个实数相加并将其结果显示在屏幕上的简单应用程序。

7.2.4.1 好的代码

为了了解创建协处理器汇编语言程序的基本情况，检查如下的程序清单：

```
// RealAdd.cpp
// Assembly Language Program
// illustrates the use of the coprocessor in
// assembly language to perform real number
// arithmetic.
// contains no errors
#include <iostream.h>
void main(void)
{
    double numa = 128.76;
    double numb = -3.5e+1;
    double answer;
    _asm {
        finit          ;initialize coprocessor
        fld     numa    ;place first real onto stack
        fadd     numb    ;add second real to stack
        fstp     answer  ;pop and save real answer
        fwait         ;sync co and micro procesors
    }
    cout << answer << "\n";
}
```

程序中以常规方式声明了三个变量，每一个变量都定义为 **double** 类型。我们将看到，这种数据类型与协处理器一起运行正常。下面是五行汇编语言代码，每一个指令都以字母 *f* 开头，其含义是这些指令是协处理器指令。

第一个指令为 **finit**，该指令通过清除内部标志和堆栈，初始化协处理器。建议在协处理器上的第一个操作永远是 **finit** 操作。



下一条指令加载一个实数到协处理器堆栈的 ST(0)位置，加载实数使用 **fld** 指令。如果第一个数是一个整数，则应该使用 **fld** 指令加载。这种区别是很重要的，因为实数实际上以编码的整数值保存在内存中。不区分 **fld** 和 **fld** 指令，则协处理器无法指出编码的实数与整数之间的区别。

fadd 指令将第二个实数添加到协处理器堆栈项 ST(0)的内容中，所以这一操作并不是向下压栈。此处使用 **fadd** 指令，可以添加一个整数。

fstp 指令将堆栈 ST(0)中的内容保存在一个变量中，并从堆栈中弹出该值。此处所保存的是一个普通编码的实数。使用 **fistp** 指令，正在保存的值可以以整数保存。

fwait 指令用于同步协处理器与微处理器的操作。程序的控制权在微处理器中，但是当微处理器检测到协处理器指令时，控制转移给协处理器。在操作返回到微处理器之前，使用 **fwait** 语句同步这些操作是非常明智的举措。

现在，编译该程序，进入 Debugger，检查该程序运行时究竟发生了什么。

图 7-18 给出了这一程序的初始调试屏幕。

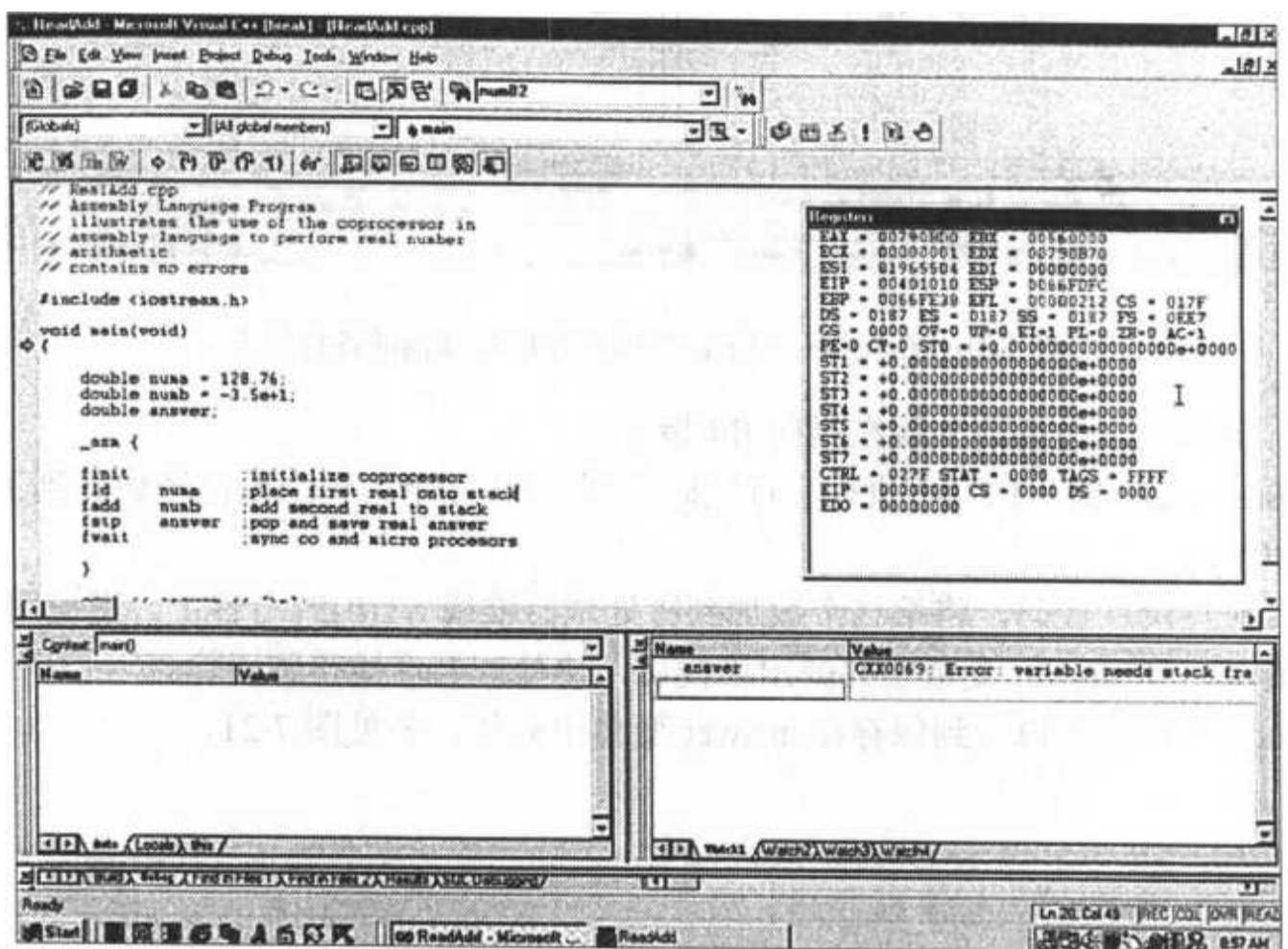


图 7-18 在 Debugger 中启动 ReadAdd.cpp 程序

Registers 窗口显示了协处理器的堆栈项，即使没有执行 **finit** 指令，各堆栈项也全都为 0。在堆栈项下面是协处理器使用的各种标志。



图 7-19 给出了执行 **finit** 指令之后协处理器值的状况。

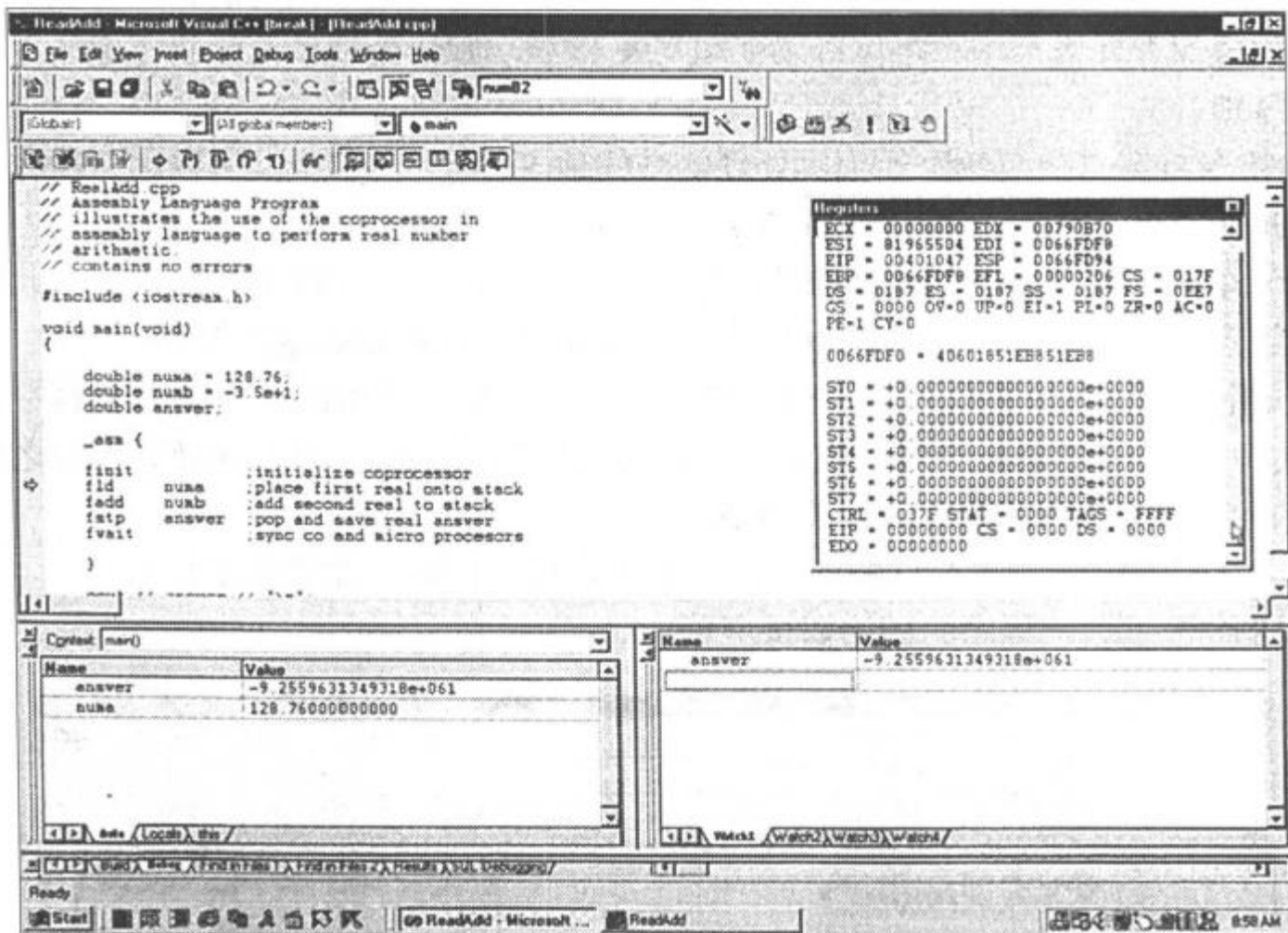


图 7-19 **finit** 指令用于清除堆栈和协处理器的各种标志

注意，屏幕最左边的箭头现在指向 **fld** 指令。

使用 **Step Into** 按钮(F11)再执行一行代码，图 7-20 显示第一个数已经加载到了协处理器堆栈的 ST(0)中。

现在，执行两行代码，将第二个数加到协处理器堆栈 ST(0)的内容上。当使用 **fstp** 指令将这一数值保存到 **answer** 变量中并弹出堆栈后，协处理器堆栈得到清除。

在 **Watch** 窗口中可以看到保存在 **answer** 变量中的值，参见图 7-21。

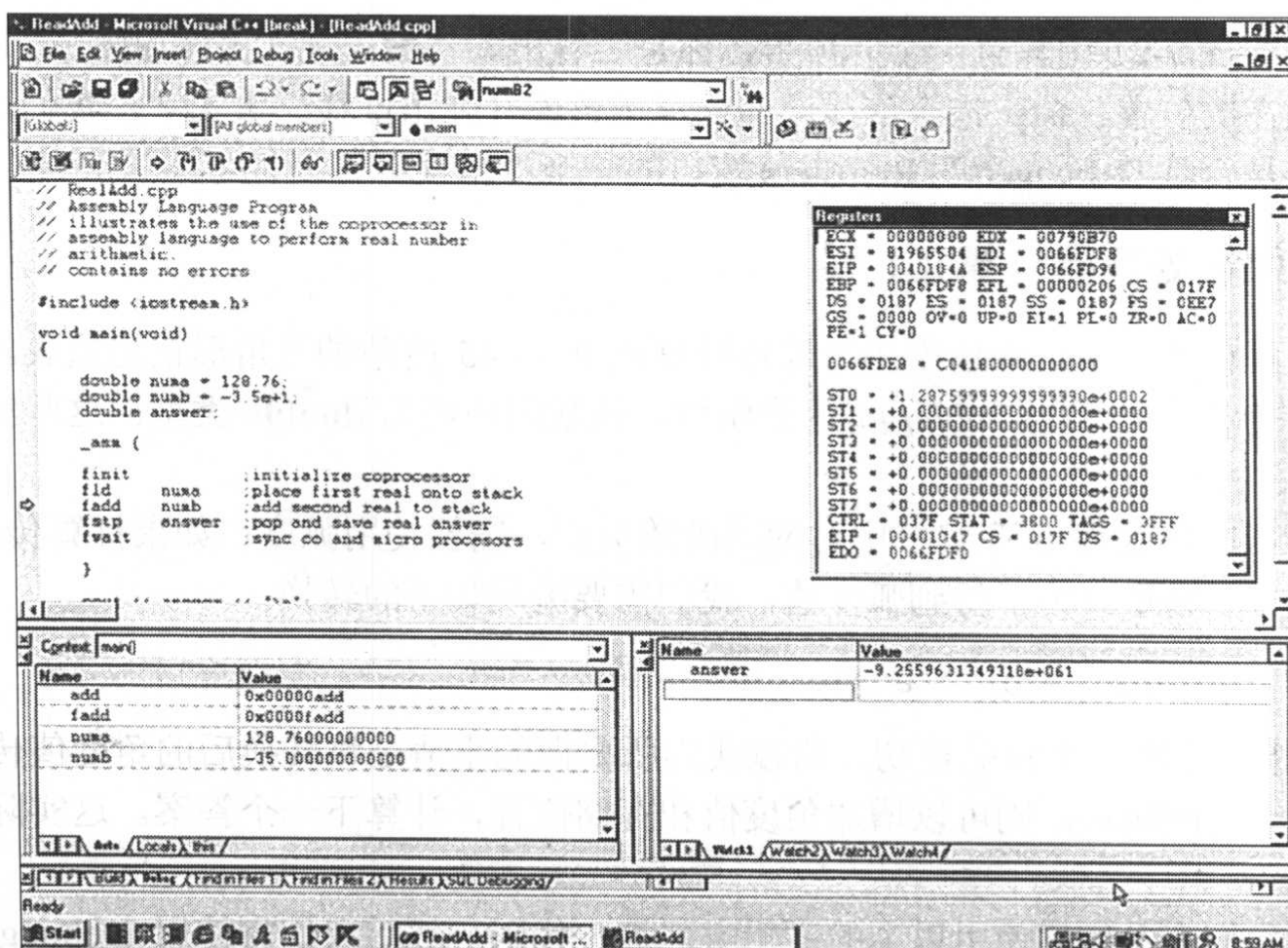


图 7-20 第一个数值加载到了协处理器的堆栈中

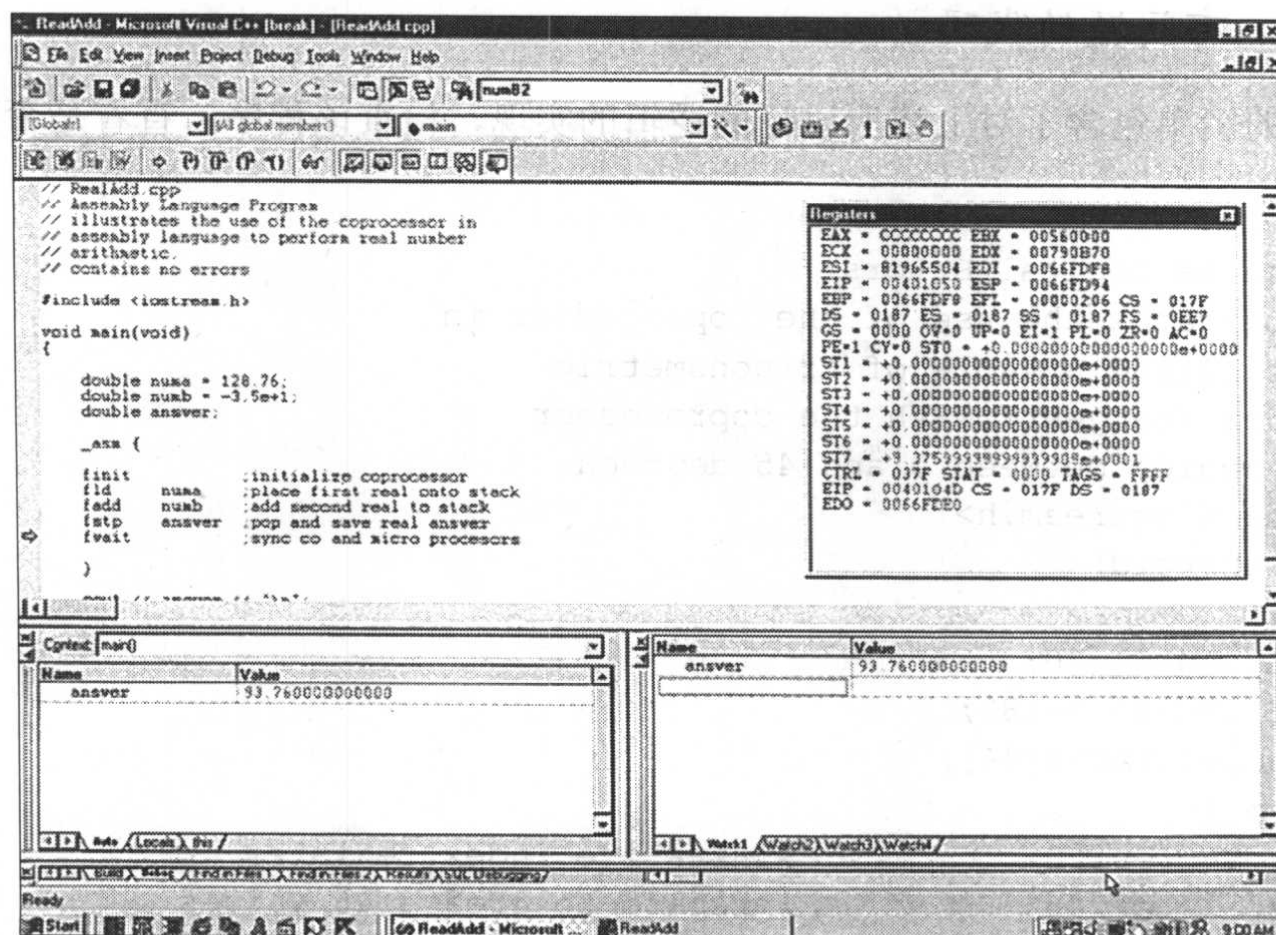


图 7-21 Debugger 的 Watch 窗口中可以看到答案



现在，在图 7-21 中只有一个麻烦事，你是否看出？答案出现在 ST(7)中了，这并不是设想中所应该在的位置。**fstp** 指令应该已经保存了该值并弹出堆栈。事实上，我们可以肯定确实如此。这是否是 Debugger 的 Registers 窗口的一个“错误”？

7.2.5 使用协处理器计算正切值

在这一程序中，我们将使用协处理器计算从 0 到 45 度角的三角形正切值表。将计算这 46 个值并保存在一个名为 *ttable[46]* 的数组中，该数组声明为 **double** 类型。这些实数的范围在 0 到 1.0 之间。

协处理器三角函数与其 C++ 中对应的函数类似，需要使用弧度。如果打算传送角度值，则必须执行一个转换。为了得到弧度值，我们需要编码如下的转换：

```
angle (radians) = pi * angle (degrees) / 180
```

fptan 指令返回一个角的正切，所以我们所要做的事情是将转换后的角度值传送给该函数。如果这一编码成功，则可以增加角度值和存储位置，计算下一个答案。这实际上是一个循环。

现在，我们查看如何使用以上的逻辑实现这一代码，然后我们将使用 Debugger 查找出所引入的逻辑错误。

7.2.5.1 有问题的代码

本节中的清单包含了对上述所讨论的逻辑的实现，检查该程序，查看是否可以立即发现问题。

```
// Contains Logical Errors
// illustrates the use of the coprocessor in
// calculating a table of trigonometric
// values (tangent) with the coprocessor
// for angles between 0 and 45 degrees.
#include <iostream.h>
void main(void)
{
    int angle = 0;
    int radian = 180;
    double ttable[46];
    _asm {
        mov     esi,0           ;initialize index register
        mov     cx,46           ;prepare to create 46 values
        finit                    ;initialize coprocessor
        more: fld     radian      ;place radian on stack
```



```

fldpi          ;load pi on stack
fdiv           ;divide pi by radian
fild    angle  ;load angle onto stack
fmul           ;convert degrees to radians
fptan          ;calculate tangent
fstp    ttable[esi] ;save answer in tangent table
inc    angle    ;next angle
add    esi,08d  ;point to next storage location
loop   more     ;end if cx=0
fwait                ;sync co and micro processors
}
cout << "Angle Tangent\n\n";
for(int i = 0; i < 46 ; i++) {
    cout << i << "\t" << ttable[i] << "\n";
}
}

```

该程序中的错误是协处理器错误，并且没有包含表索引或循环记数。

从 **finit** 指令开始的协处理器的前六行代码，使用前面所介绍的公式，将角度值转换为弧度值。

fptan 指令使用这一转换了的角度计算并返回一个角的正切，然后使用 **fstp** 指令将实数结果返回到 *ttable[]* 数组中。

编译后执行该代码，检查相应窗口中的显示内容。

图 7-22 中给出了 22 到 45 度角之间的结果。

正如我们所怀疑的那样，这一结果甚至十分荒谬。现在，启动 **Debugger** 后查找这一编程错误。

启动 **Debugger**，打开 **Registers** 窗口，将 *ttable* 添加到 **Watch** 窗口中，单步执行代码到 **fldpi** 指令，如图 7-23 所示。

检查 **ST(0)**，可以看出该常量已经正确加载到了堆栈中。现在执行另一行代码，图 7-24 显示了所执行的下一行代码为 **fdiv** 指令，由屏幕左边的箭头位置指定。

再次检查 **ST(0)**，现在其中包含了一个精确的 *pi* 值，还注意到了压到堆栈 **ST(1)** 中的常量值。通过运行下一行代码，执行一个除法运算，如图 7-25 所示。

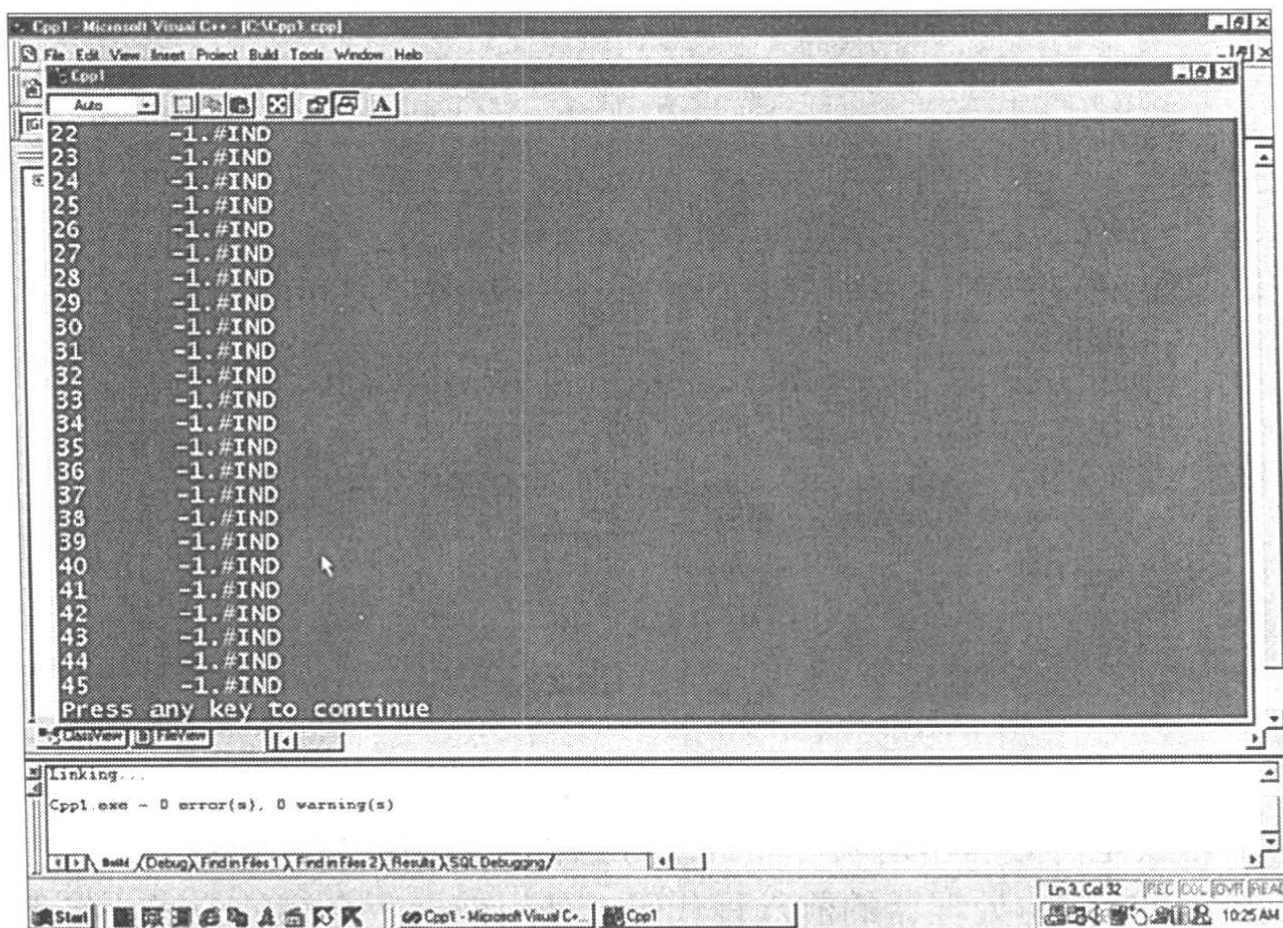


图 7-22 是 0 到 45 度角之间的正切值否?



图 7-23 检查弧度常量是否正确加载

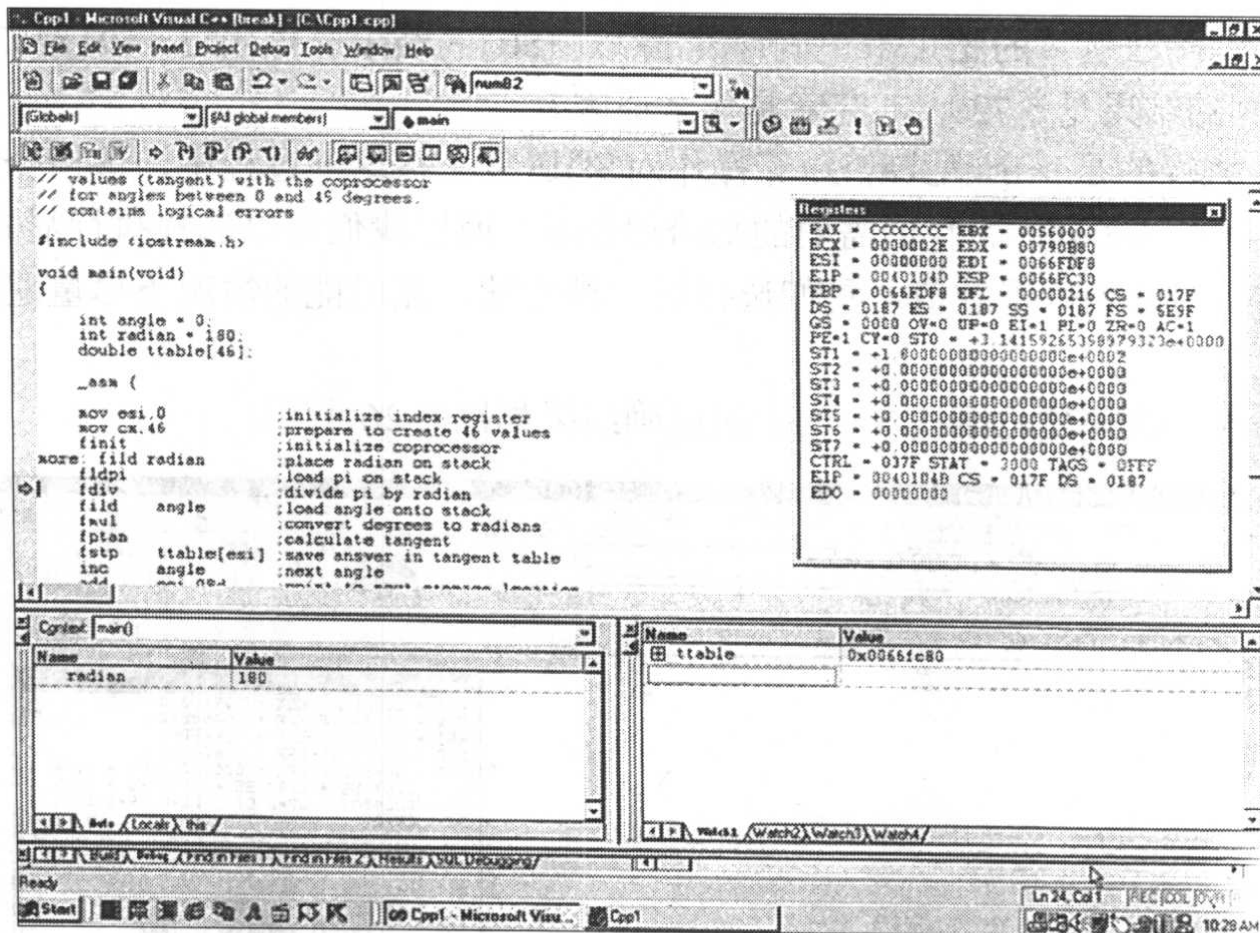


图 7-24 检查 pi 值是否正确加载

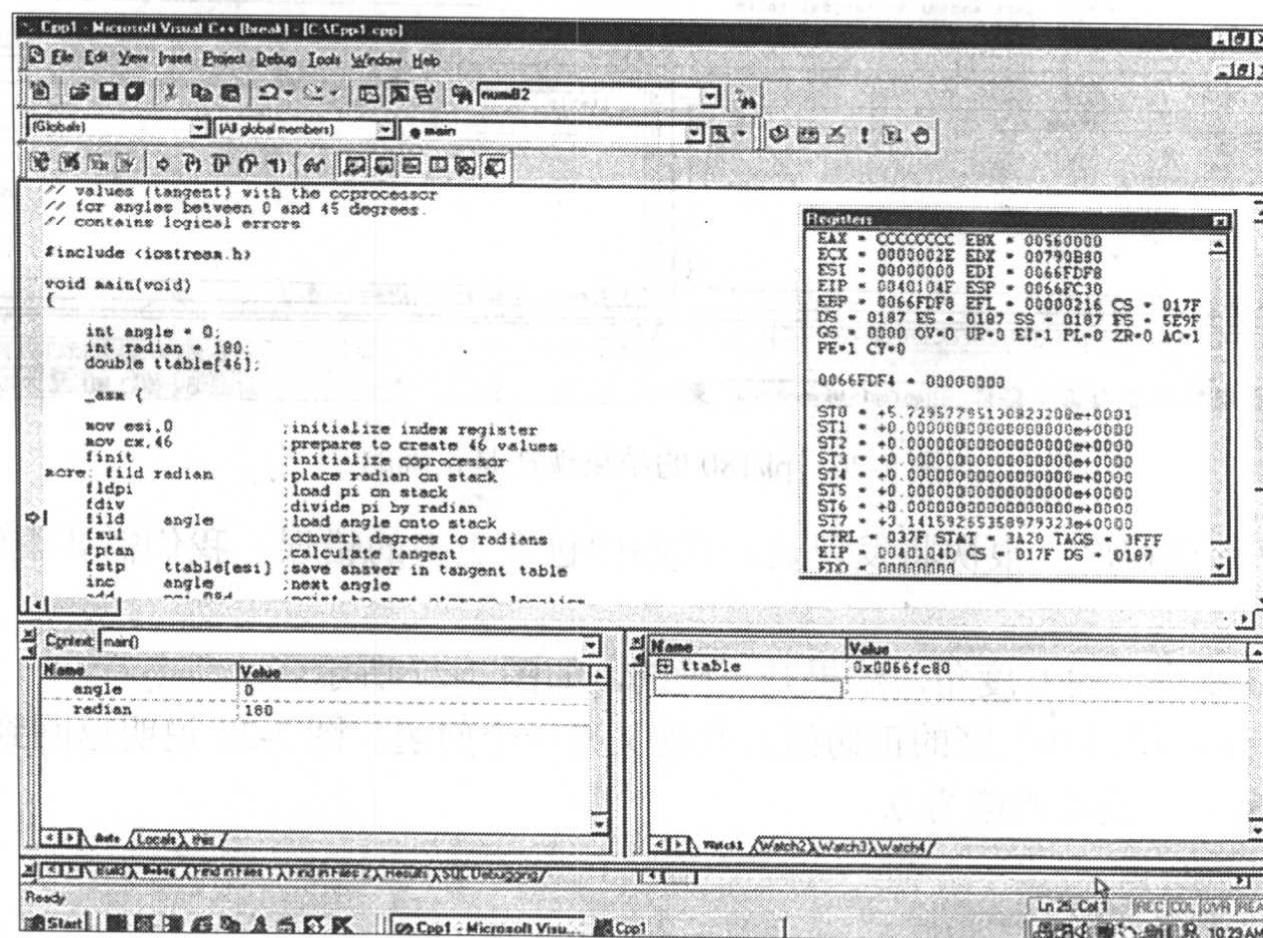


图 7-25 检查除法操作(pi/180)



可以看出，除法运算的结果不正确。 π 除以 180 的结果应该接近于 0.017453...，但我们在 ST(0) 中得到的结果是 5.729...，究竟是什么原因？

fdiv 指令执行的是一个隐式除法(无操作对象)运算。在这一过程中，该指令执行的除法为 ST(1)/ST(0)，即 $180/\pi$ 。而我们需要的是恰好相反，所以我们可以选择执行显式除法运算，或重新放置堆栈中开始位置的值。我们选择后一种方法，在可能的情况下尽量避免显式的协处理器运算操作。

检查代码的修改部分，堆栈 ST(0) 中的正确结果如图 7-26 所示。

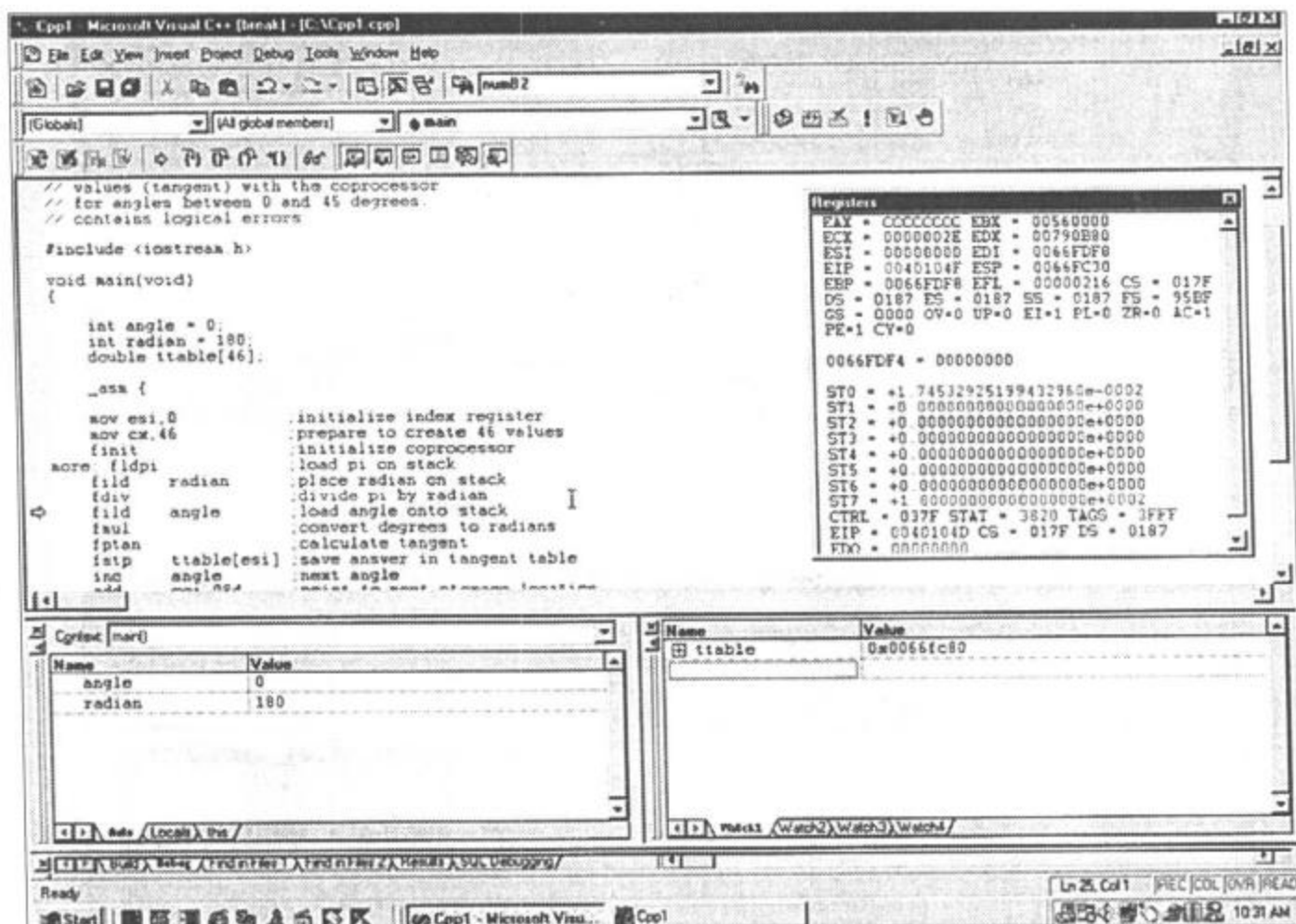


图 7-26 $\pi/180$ 的结果现在是正确的

如果再执行该程序，很快将发现这一代码中包含更多的错误。我们需要继续调试这一程序。

执行 **fmul** 指令，对 0 度角，结果是正确的，如图 7-27 所示。

再执行一步，找出 0 弧度的正切值。此处又有一个问题。图 7-28 说明返回到 ST(0) 的值为 1，而其他所有堆栈项的值为 0。

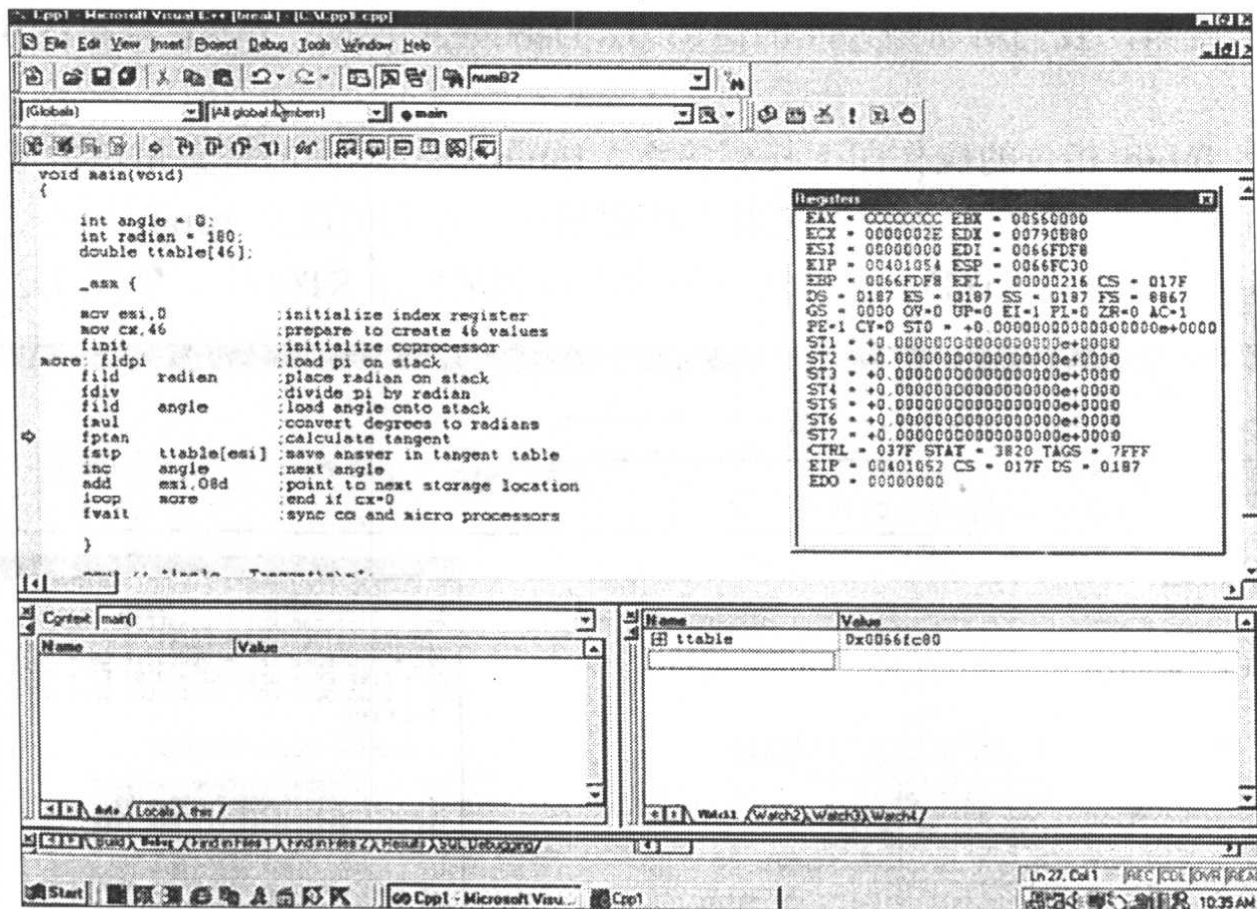


图 7-27 角度 0 正确地转换为弧度



图 7-28 fptan 指令执行后返回的结果不正确



对于 0 弧度角，我们所希望返回的值为 0。Debugger 说明，**fptan** 指令的执行存在一个问题。

稍加分析 **fptan** 指令的操作即可找到答案。**fptan** 指令返回的是直角三角形中，以 x 和 y 值表示的角的正切。现在，问题看来并不是很糟糕。我们可以在 **fptan** 指令之后，使用一个 **fdiv** 指令用 $ST(0)$ 除以 $ST(1)$ 。正确的结果将返回到堆栈顶 $ST(0)$ 中，如图 7-29 所示。

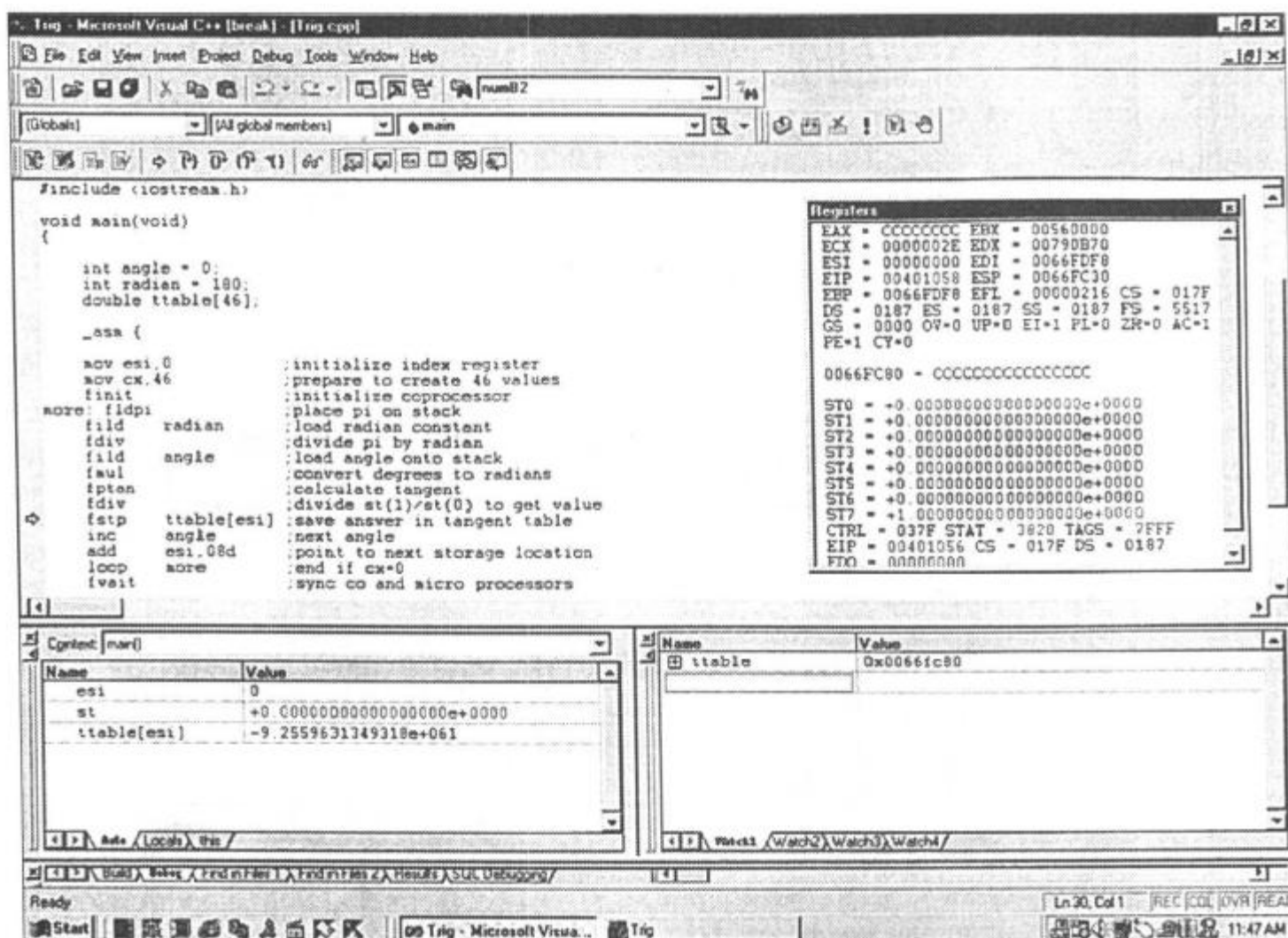


图 7-29 对角度 0 的正切返回结果正确

进一步检查，单步执行若干个角度，并注意堆栈中的结果。图 7-30 给出了 $ST(0)$ 和 $ST(1)$ 项中，对于 20 度角的情况。

很显然，我们已经找出并修复了该程序中的所有错误。可以继续对其余的角度单步执行，检查其结果。



图 7-30 对角度 20 的正切返回结果正确

7.2.5.2 好的代码

纠正了所有的错误后，请看如下的正确程序，现在将其命名为 Trig.cpp。

```
// Trig.cpp
// Assembly Language Program
// illustrates the use of the coprocessor in
// calculating a table of trigonometric
// values (tangent) with the coprocessor
// for angles between 0 and 45 degrees.
// contains no errors
#include <iostream.h>
void main(void)
{
    int angle = 0;
    int radian = 180;
    double ttable[46];
    _asm {
        mov esi,0                ;initialize index register
```



```

        mov cx,46                ;prepare to create 46 values
        finit                    ;initialize coprocessor
more: fldpi                      ;place pi on stack
        fild    radian          ;load radian constant
        fdiv    ;divide pi by radian
        fild    angle          ;load angle onto stack
        fmul    ;convert degrees to radians
        fptan   ;calculate tangent
        fdiv    ;divide st(1)/st(0) to get value
        fstp    ttable[esi]    ;save answer in tangent table
        inc     angle          ;next angle
        add     esi,08d        ;point to next storage location
        loop    more           ;end if cx=0
        fwait   ;sync co and micro processors

    }
    cout << "Angle  Tangent\n\n";
    for(int i = 0; i<46 ; i++) {
        cout << i << "\t" << ttable[i] << "\n";
    }
}

```

24x7

从历史上看，**fptan** 指令是第一个协处理器(8087)中可以使用的少数几个三角函数之一。这种返回直角三角形的 x 和 y 值的安排，使得可以计算所有其他的三角函数值，如正弦、余弦、正割、余割、余切等。**fptan** 指令也受到限制，即其只能在允许的角度范围内使用。基本上说，**fptan** 可以精确返回 0 到 45 度范围内的值。对于以后的处理器，**fptan** 函数开始接收全部范围内的角度值，并相应提供了 **fsin** 和 **fcos** 指令。然而在编写代码时，一定要记住，**fptan** 并不直接返回正切值。正切值必须由返回到协处理器堆栈中的 x 和 y 值计算而来。

编译并执行该程序，其结果显示在图 7-31 中。

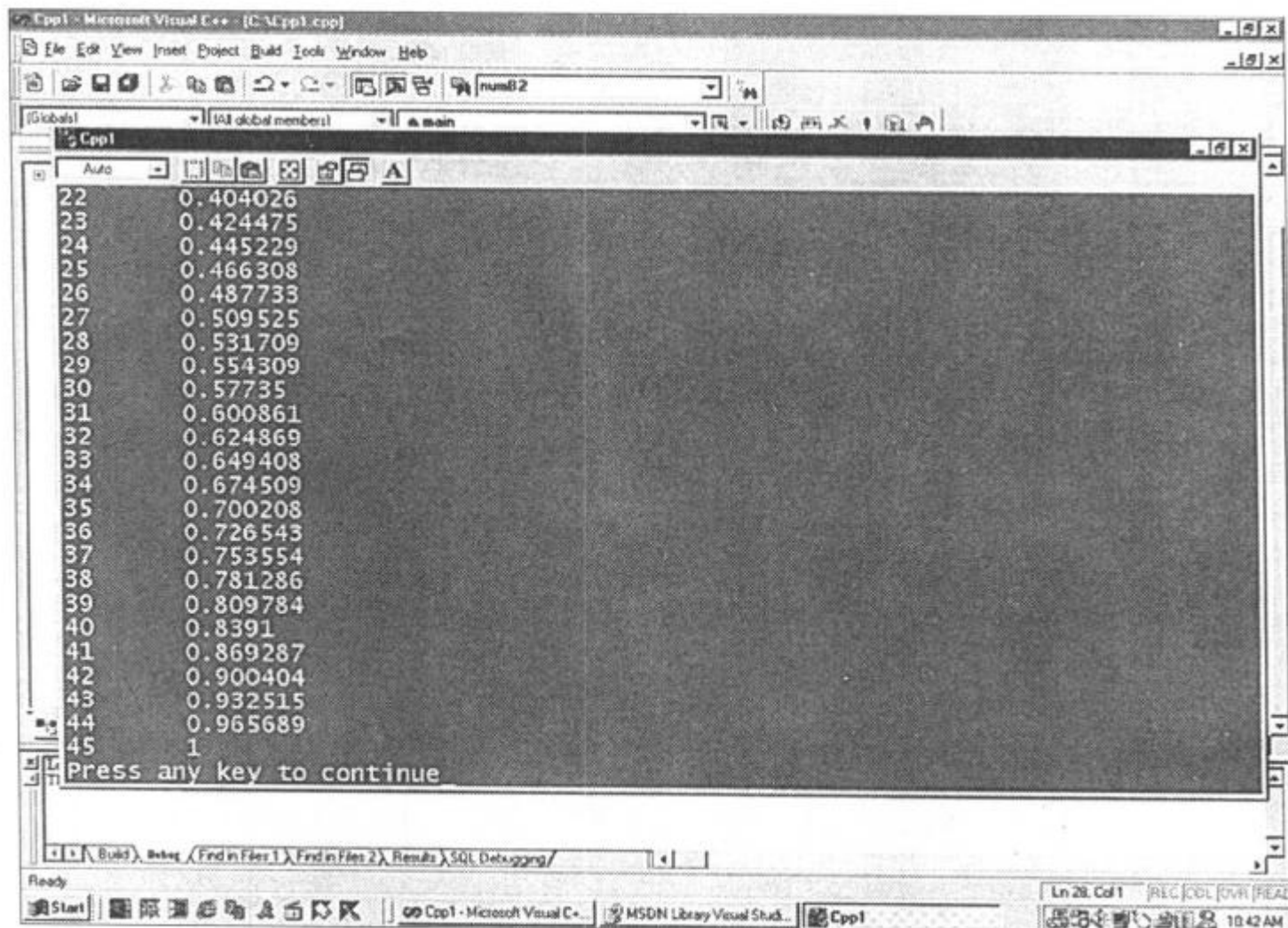


图 7-31 一个正确计算和格式化的三角值表

读者可能需要使用该程序做一个试验,为什么不可以对角度 0 到 90 度计算一个正弦表?

设计提示

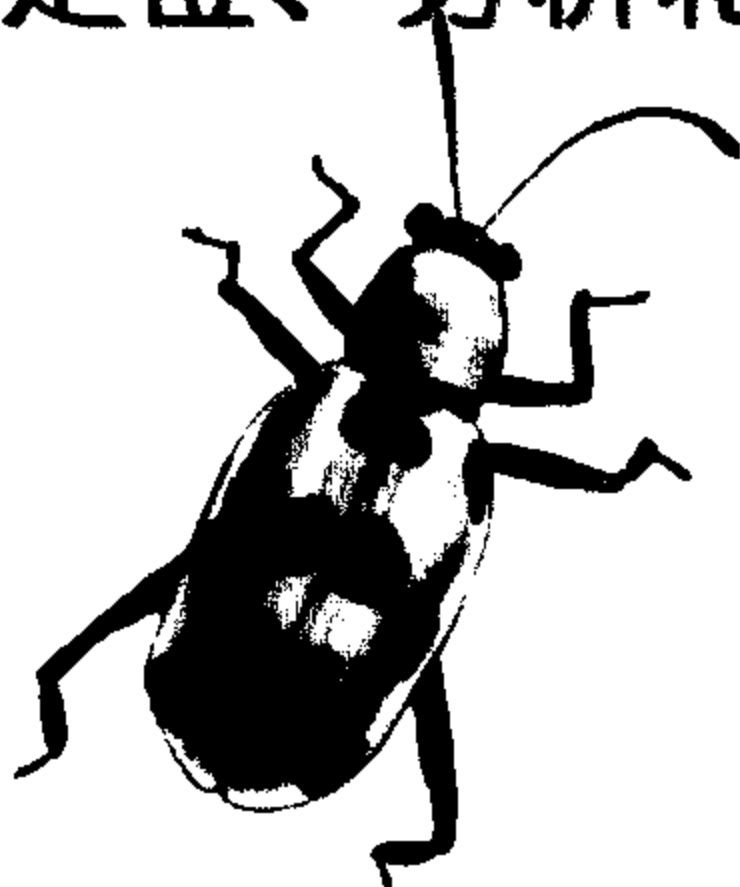
在使用协处理器指令如 **fsin** 和 **fcos** 时,不需要在操作之后执行一个隐式 **fddiv**, 这些函数将直接返回正确的值并可以作用于更大范围的角度。

7.3 小结

本章介绍了一些功能强大的调试技术,包括微处理器和协处理器的使用。我们学习了如何在 Debugger 的 Registers 窗口中查看微处理器寄存器和协处理器堆栈项的方法,这些信息可用于跟踪和消除各种汇编语言的逻辑错误。 ■

第 8 章

在 Windows 代码中
定位、分析和修复错误





本章内容建立于读者对于面向过程的 Windows 应用程序开发的理解和前面各章中所提供的调试资料的基础之上。本章中用于说明各种调试技术的代码，可以在只做少量修改或不做任何修改的情况下编译运行于 Windows 98 和 Windows NT(2000)操作系统下。

大部分的 Windows 应用程序都是由 C 或 C++编写的，Visual Basic 只是作为一种密切配合的第二语言。正如命令行的 C++程序可以是面向过程的或面向对象的一样，使用 C++编写的 Windows 应用程序也可以是面向过程的或面向对象的。本章将重点讨论完全是面向过程的方法，第 10 章到第 11 章和第 14 章到 17 章将集中讨论功能强大的 C++面向对象的环境。

读者在开发 Windows 应用程序时是否使用过面向过程或面向对象的环境？这确实是一个复杂的问题，并且该问题的答案要依赖于若干个方面。让我们做这样的一种假设：如果程序较为简单并使用了适中的资源，那么可以看出面向过程的环境即足以满足需要。实际上，与面向对象的同样的程序相比，编译后的代码长度可能更小一些，应用程序的运行速度可能更快一些。

然而，无论是否喜欢开发面向过程的代码，也无论多么地不愿意接触除了面向对象的应用程序之外的内容，都应该阅读本章内容。应该学习关于 Debugger 和关于调试的内容，这将有助于本书以后各章的学习。

8.1 使用两台计算机调试

警告，在将早期的第二个计算机系统送给 Aunt Annetta 以寻求其电子邮件之前，请首先阅读本节内容！

早期的计算机用户常常感到奇怪，为什么有些人要使用多媒体环境。运行一个应用程序不只是满足其需求。早期多任务方面的先驱，如 TopView 和 DeskView，已经为 OS/2 和 Windows 铺平了道路。作为用户，我们已经非常适应于同时运行多个应用程序。即使在笔者编写本章内容时，我们的计算机上也正在运行着 Microsoft Word、Microsoft Visual C++以及 Collage(一个屏幕捕获程序)。

本节中我们将更进一步，在 Debugger 中运行一个 Windows 应用程序，并且最好是使用两台计算机。一台计算机可以显示 Windows 应用程序，另一台计算机用来显示 Debugger 的内容。

问题集中在了 Windows 应用程序的本质——一个频繁运行于前台并且保持焦点的应用程序(即位于屏幕的最上层并接收键盘和鼠标输入)。然而，Debugger 也是一个 Windows 应用程序，它也需要焦点。解决问题的方案是使用两台网络计算机、两个键盘、两个鼠标和两台监视器。Windows 应用程序将自己使用一个系统，Debugger 使用另一个系统。所有这些都是通过网络连接 Windows 应用程序与 Debugger 的一种手段。



Microsoft 提供了对这一问题的解决方案，可以从 Visual Studio 提供的 MSDN Library 中找到建立计算机之间通信的纲要，只要查看“Debugging Remote Applications”主题即可。在下一节中，我们将介绍使用两台计算机调试的各个步骤，并依次给出具体建议。重要的是两台计算机之间必须使用 TCP/IP 实现网络连接。下一节中所使用的这些步骤是针对 Visual C++ 6.0 的，如果读者正在使用的是更低或更高的版本，可能需要做一些修改。

建立这种连接时，术语是成功的关键。Microsoft 将主计算机系统称为主机(host)，远程计算机系统称为目标机(target)。在笔者的环境中，主计算机为一台 600MHz 的 Hewlett-Packard Pentium III 计算机，而目标计算机是一台 400MHz 的 Sony Vaio 便携机。在网络上，Hewlett-Packard 主机使用“hp”识别，Sony Vaio 目标机使用“sony”识别。

8.1.1 准备远程目标计算机

一个称为 Remote Debug Monitor 的小程序必须运行于远程目标计算机上——在笔者的环境中为 Sony 系统。Remote Debug Monitor 应用程序和目标计算机负责与运行于主计算机(笔者的环境中为 Hewlett-Packard)上的 Debugger 通信，Remote Target Computer 软件控制应用程序在 Debugger 中的执行。

为了安装 Remote Debug Monitor，一些附加文件也必须复制到远程目标计算机(sony)上。对于 Windows 98 和 Windows NT(2000)，这些文件包括：

```
MSVCMON.EXE
MSVCRT.DLL
TLN0T.DLL
DM.DLL
MSVCP60.DLL
MSDIS110.DLL
PSAPI.DLL (仅用于 NT)
```

为了在适当的主计算机子目录中定位这些文件，只要使用系统的 Start 窗口的 Find 选项即可。应该将这些文件从主计算机(hp)上拷贝到远程目标计算机(sony)上，并将其保存在远程目标系统的 Windows 子目录下。只有一个例外，即 MSVCRT.DLL 应该复制到 Windows\System32 子目录下。当完成这些复制后，重新引导计算机。

为了在远程目标系统(sony)上运行 Remote Target Computer 软件，执行如下步骤：

1. 在远程目标计算机(sony)上运行 MSVCMON.EXE 应用程序。
2. 显示 Visual C++ Debug Monitor 对话框。
3. 选择 Settings 选项。
4. 显示 Win32 Network (TCP/IP)Settings 对话框。
5. 在该对话框中输入主计算机(hp)的名字(IP 地址也要使用)。



6. 如果口令 Edit 框为活动状态, 则输入口令, 这一口令在两台计算机上必须匹配; 否则该域可以保留为空。

7. 单击 OK 按钮。

8. 单击 Connect 按钮。

当完成如上的 8 个步骤后, 将出现 Connecting 对话框, 如图 8-1 所示。

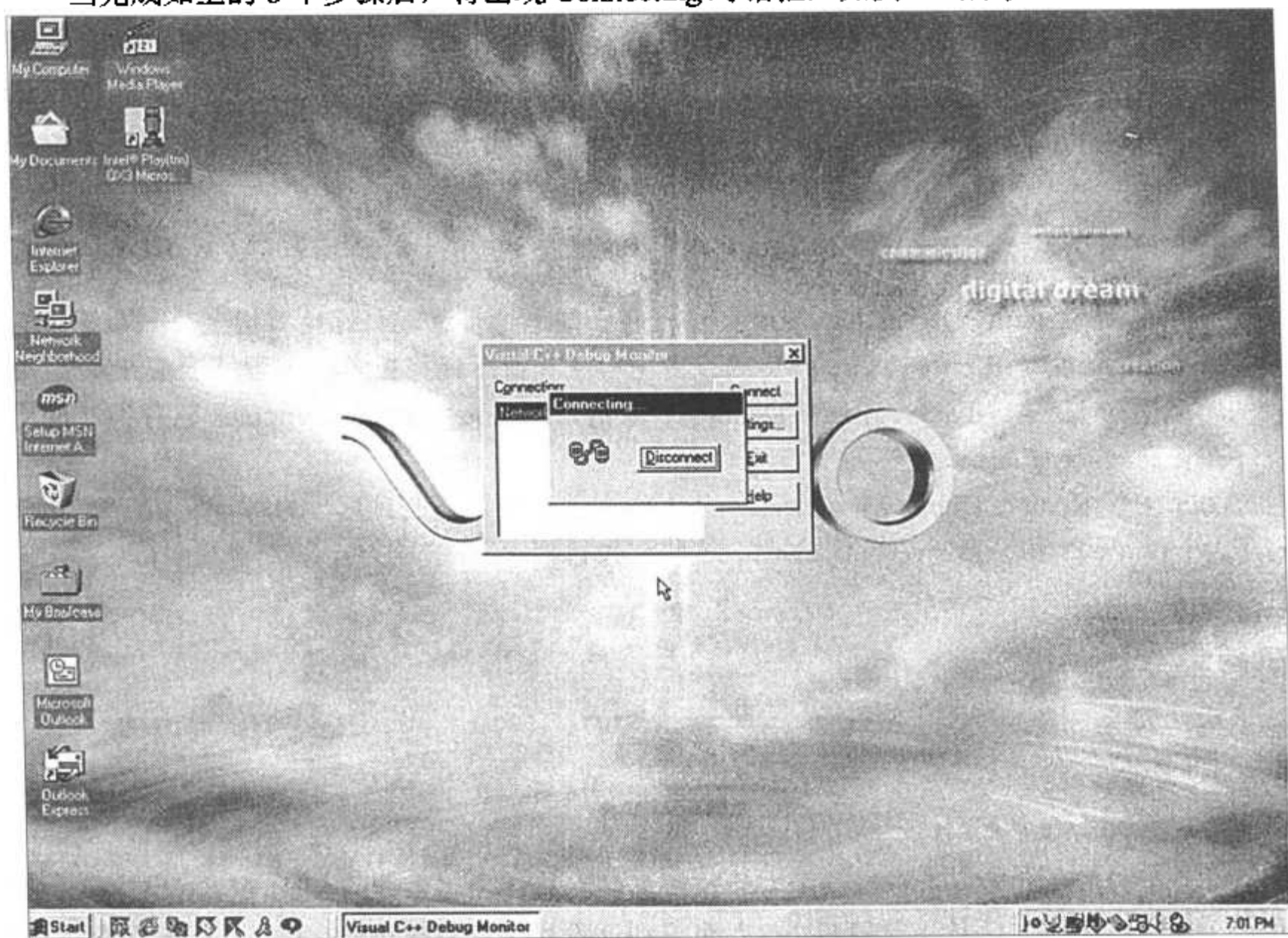


图 8-1 远程目标计算机上出现的 Connecting 对话框

此时不需要做什么工作, 当实际调试开始后该对话框将消失。

当调试会话完成后, 选择 Disconnect 按钮结束远程连接。

8.1.2 准备主计算机

必须为与远程目标计算机(笔者的环境中为 sony)通信而准备主计算机(笔者的环境中为 hp), 在主计算机(hp)上执行如下步骤:

1. 在 Visual C++ 中选择 Build|Debugger Remote Connection 菜单项。

2. 显示 Remote Connection 对话框。
3. 如果 Platform 下拉列表框允许选择，则选择适当的平台：如果没有提供可选择的选项，则将自动选择缺省项。
4. 使用 Connection 下拉列表框选择 Network(TCP/IP)连接选项。
5. 现在，选择 Settings 选项。
6. Win32 Network(TCP/IP)Settings 对话框将出现。
7. 在该对话框中输入远程目标计算机的名字(笔者的环境中为 sony)，IP 地址也要用到。如果口令选项可以使用，则输入与远程目标的口令相同的口令。这一域也可以在主机和远程机器上同时保持为空。
8. 单击 OK 按钮，关闭 Win32 Network(TCP/IP)Settings 对话框。
9. 单击 OK 按钮，关闭 Remote Connection 对话框。

图 8-2 给出了主计算机(hp)上的 Remote Connection 和 Win32 Network Settings 对话框。

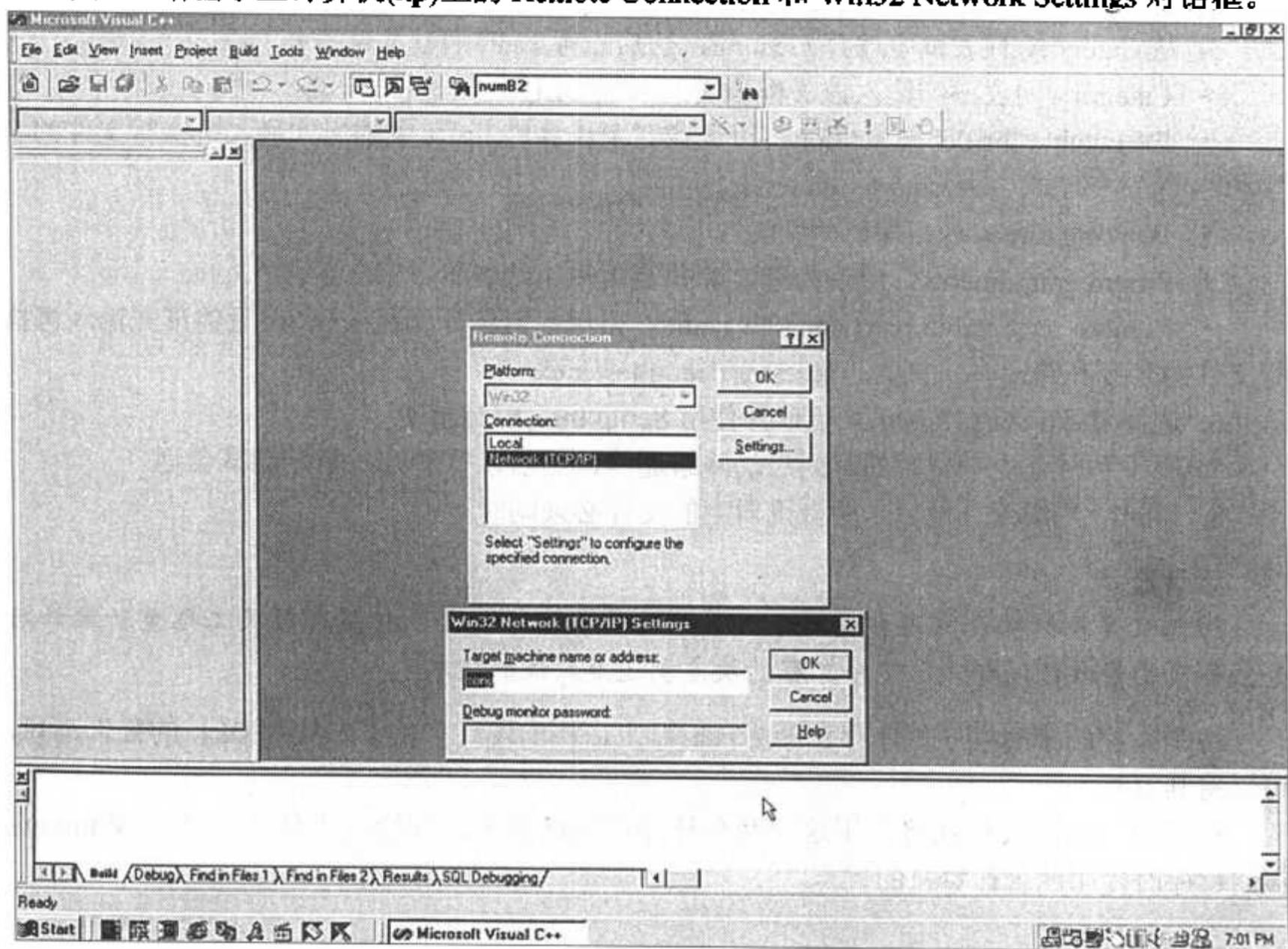


图 8-2 主计算机上的 Remote Connection 和 Win32 Network Settings 对话框



当以上的 9 步完成之后，主计算机即做好了与远程目标计算机通信的准备。

8.1.3 启动调试会话

当网络上的两台计算机都完成通信准备后，则可以启动调试会话(切记，在笔者的系统中运行 Visual C++ 的主计算机为 Hewlett-Packard，远程目标计算机为 Sony)。执行如下步骤：

1. 将要调试的完整工程复制到两台机器上。笔者建议使用相同的目录，例如，复制文件应存在于两台计算机的 `c:\myproject` 目录下，并且调试的可执行程序在 `c:\myproject\debug` 目录下。两个目录都应该设置为网络共享目录。

2. 启动 Visual C++ 编译器，并加载该工程到主计算机(远程目标计算机(sony)上不需要 Visual C++ 编译器)。

3. 选择 Project|Settings 菜单选项，Project Settings 对话框出现。

4. 选择 Project Settings 对话框中的 Debug 标签。

5. 从 Category 列表框中选择 General，然后设置如下项目(参见图 8-3)：

- Category 列表框：输入该工程所需的附加 DLL。
- Executable for debug session：以调试器主计算机(hp)可见的形式输入可执行文件的名字和路径——例如，`c:\tester\debug\tester.exe`。
- Working directory：保持为空。
- Program arguments：保持为空，除非要接收初始参数。
- Remote executable path and file name：以远程目标计算机(sony)可见的形式输入可执行文件的名字和路径——例如，`c:\tester\debug\tester.exe`。

6. 选择 Build|Start Debug 菜单项并使用 Setup Info(F11)选项。

7. 给予系统至少 1 分钟的时间启动网络通信，然后以普通方式开始调试会话。

8. 切记，当修改工程后，两台机器上的文件必须同时更新。

设计提示

将工程复制到两台机器相同的子目录下可能有些不必要，但是与每次工程重新编译后查找各自的修改情况相比，整个工程的块复制还是最快的技术。

由于完整的应用程序复件存在于两台机器上，将出现许多关于未找到 DLL 的警告消息，可以将其忽略。

在本章的后面我们还将使用这一两台计算机调试技术，并给出为什么在调试 Windows 应用程序时使用两台计算机的优点。

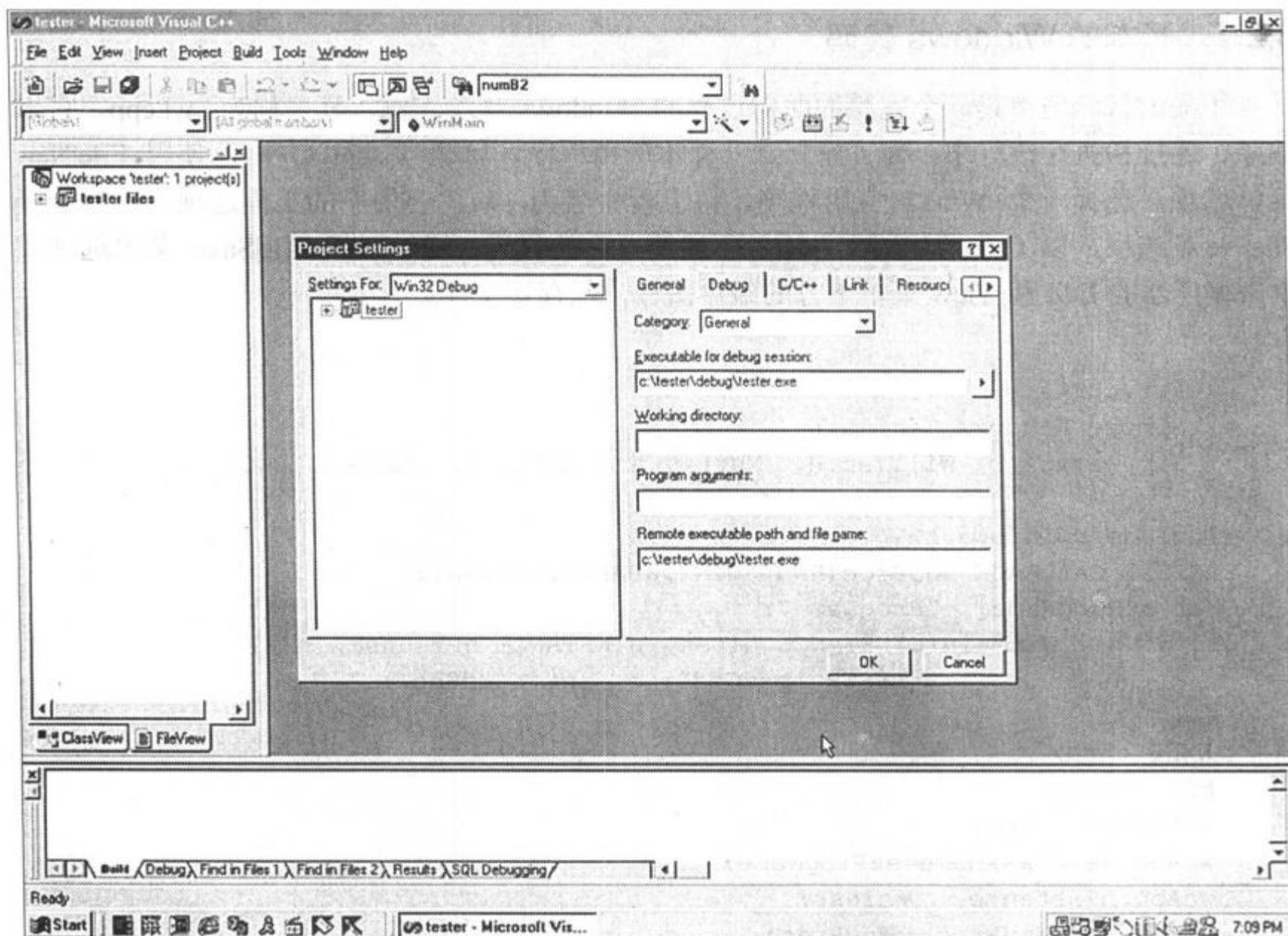


图 8-3 出现在主计算机(hp)上的 Project Settings 对话框

8.2 简明 Windows 入门

本节中我们将检查在屏幕上创建一个简单的窗口并在窗口中绘制一些简单图形所需的代码。在给出的代码之后，将是关于所使用的面向过程的 Windows 函数的重要解释。如果读者还没有使用过 Windows 环境，那么笔者强烈推荐首先使用一本关于该主题的专门书籍，自己熟悉这方面的内容。当然，笔者建议使用由笔者编写，由 Prentice Hall 推出的《*Introduction to Windows 98 Programming*》一书。本节中我们并不打算完全重复关于 Windows 编程的多达 500 到 800 页的一本书，而是作为本章后面将提供的调试材料的预备知识，给出关于 Windows 应用程序基本结构的简要说明。



8.2.1 基本的 Windows 代码

下面的代码清单是一个完整的面向过程的 Windows 程序实例，其名称为 swt.cpp，可以编译、连接和执行该程序。为了建立这一样本应用程序，启动 Visual C++，使用 File|New 菜单选项，选择一个 Win32 应用程序。将工程命名为 swt，使用空的工程选项。再次选择 File|New 选项，以 C++ 源代码文件添加下述清单中的代码。通过选择 File|Save 菜单选项或单击编译器的 Edit 和 View 菜单下方的磁盘图标，保存该文件。

```
//  
// swt.cpp  
// Simple Windows Template  
// Copyright (c) William H. Murray and Chris H. Pappas, 2000  
//  
#include <windows.h>  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
Char szProgName[]="ProgName";  
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,  
                  LPSTR lpszCmdLine, int nCmdShow)  
{  
    HWND hWnd;  
    MSG lpMsg;  
    WNDCLASS wcApp;  
    wcApp.lpszClassName=szProgName;  
    wcApp.hInstance    =hInst;  
    wcApp.lpfnWndProc  =WndProc;  
    wcApp.hCursor      =LoadCursor(NULL, IDC_ARROW);  
    wcApp.hIcon         =0;  
    wcApp.lpszMenuName =0;  
    wcApp.hbrBackground=(HBRUSH) GetStockObject(WHITE_BRUSH);  
    wcApp.style         =CS_HREDRAW|CS_VREDRAW;  
    wcApp.cbClsExtra    =0;  
    wcApp.cbWndExtra    =0;  
    If (!RegisterClass (&wcApp))  
        Return 0;  
    hWnd=CreateWindow(szProgName,"Simple Windows Template",  
                     WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,  
                     CW_USEDEFAULT,CW_USEDEFAULT,  
                     CW_USEDEFAULT, (HWND) NULL, (HMENU) NULL,  
                     hInst, (LPSTR) NULL);  
    ShowWindow(hWnd, nCmdShow);  
    UpdateWindow(hWnd);  
    while (GetMessage(&lpMsg, 0, 0, 0)) {
```




```

        TranslateMessage(&lpMsg);
        DispatchMessage(&lpMsg);
    }
    return(lpMsg.wParam);
}
LRESULT CALLBACK WndProc(HWND hWnd,UINT messg,
                        WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    switch (messg)
    {
        case WM_PAINT:
            hdc=BeginPaint(hWnd,&ps);
            MoveToEx(hdc,45,55,NULL);
            LineTo(hdc,480,410);
            TextOut(hdc,200,100,"A simple line",13);
            ValidateRect(hWnd,NULL);
            EndPaint(hWnd,&ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return(DefWindowProc(hWnd,messg,wParam,lParam));
            break;
    }
    return(0);
}

```

使用 Visual C++ 的 Build/Rebuild All 菜单项编译后创建一个带有调试信息的可执行程序，Debug 子目录将保存调试信息和调试版本的应用程序执行代码 swt.exe。然而还可以看到大量的其他附加文件，其中有些是很重要的。例如 swt.cpp 文件是刚刚输入后保存的 C++ 源文件，swt.dsp 文件是开发环境内使用的工程文件，swt.dsw 文件也是开发环境内使用的工作空间文件，swt.ncb 文件是 No 编译浏览器文件，其中包含了 Visual Studio 工具，如 Class View、WizardBar 及 Component Gallery 等所使用的语法分析程序所产生的信息。

8.2.2 调试文件详述

在通过执行 Rebuild All 构造一个 C++ 工程时，Visual Studio 产生了一组文件，并允许程序员执行和调试其 Windows 应用程序。

包含在工程的 Debug 子目录中的文件之一是 swt.exe。swt.exe 文件是 swt.cpp 的机器语



言可执行版，然而该子目录下的其他文件的目的并非显而易见的。

swt.ilc 文件是执行增量连接时使用，此处，LINK 将更新首次增量连接时所创建的.ilc 状态文件。该文件与.exe 或.dll 文件有相同的基本名，其扩展名为.ilc。在以后的增量连接中，LINK 将更新.ilc 文件。如果找不到.ilc 文件，则 LINK 执行一个完全连接，并创建一个新文件。如果.ilc 文件不能使用，则 LINK 将执行一个非增量连接。

swt.obj 文件是有编译器产生的一个中间文件，并作为连接器的输入使用。

swt.pch 文件是一个预编译的头文件。通过预编译工程所使用的各种头文件，随后的构造速度将更快。

swt.pdb 文件(程序数据库)保存调试和工程状态信息，.pdb 文件为调试版本程序提供增量连接所需的信息。对于 32 位的可执行程序，连接器和集成 Debugger 均允许在调试过程中直接使用.pdb 文件，以减少连接器的巨大工作量，并避免 CVPACK 64K 类型限制的麻烦。缺省情况下，当构造由 Visual C++产生的工程时，要使用编译器开关/Fd 将.pdb 文件重命名为 project.pdb，所以在整个工程中只有一个.pdb 文件。

Vc60.idb 文件使用 Enable Incremental Compilation(Gi)选项，控制增量编译。只编译那些自上次编译以来修改过的函数。编译器将自第一次编译开始的状态信息保存在工程的.idb 文件中(缺省名称为 project.idb，对于没有工程而编译的文件为 VC50.idb)，编译器使用这些状态信息加速随后的编译。

当启动了一个实际的调试会话时，将使用到许多如上的特性。

8.2.3 程序执行的情况

让我们检查 swt.exe 文件实际执行时所发生的情况，当运行该程序时应该看到一个类似于图 8-4 所示的新窗口。

swt.cpp Win32 Application 不仅产生一个初始的窗口框架、程序标题栏和最小化、最大化及关闭图标，而且画出了一个带有文本标签的斜线。源代码的哪些部分负责处理这些组件？如下简要说明每个组件。

8.2.3.1 注释块

注释块给出了文件的名称、标题、目的、作者以及创建/修改的日期。注释块是可选的，但非常重要！

```
//  
// swt.cpp  
// Simple Windows Template  
// Copyright (c) William H. Murray and Chris H. Pappas, 2000  
//
```

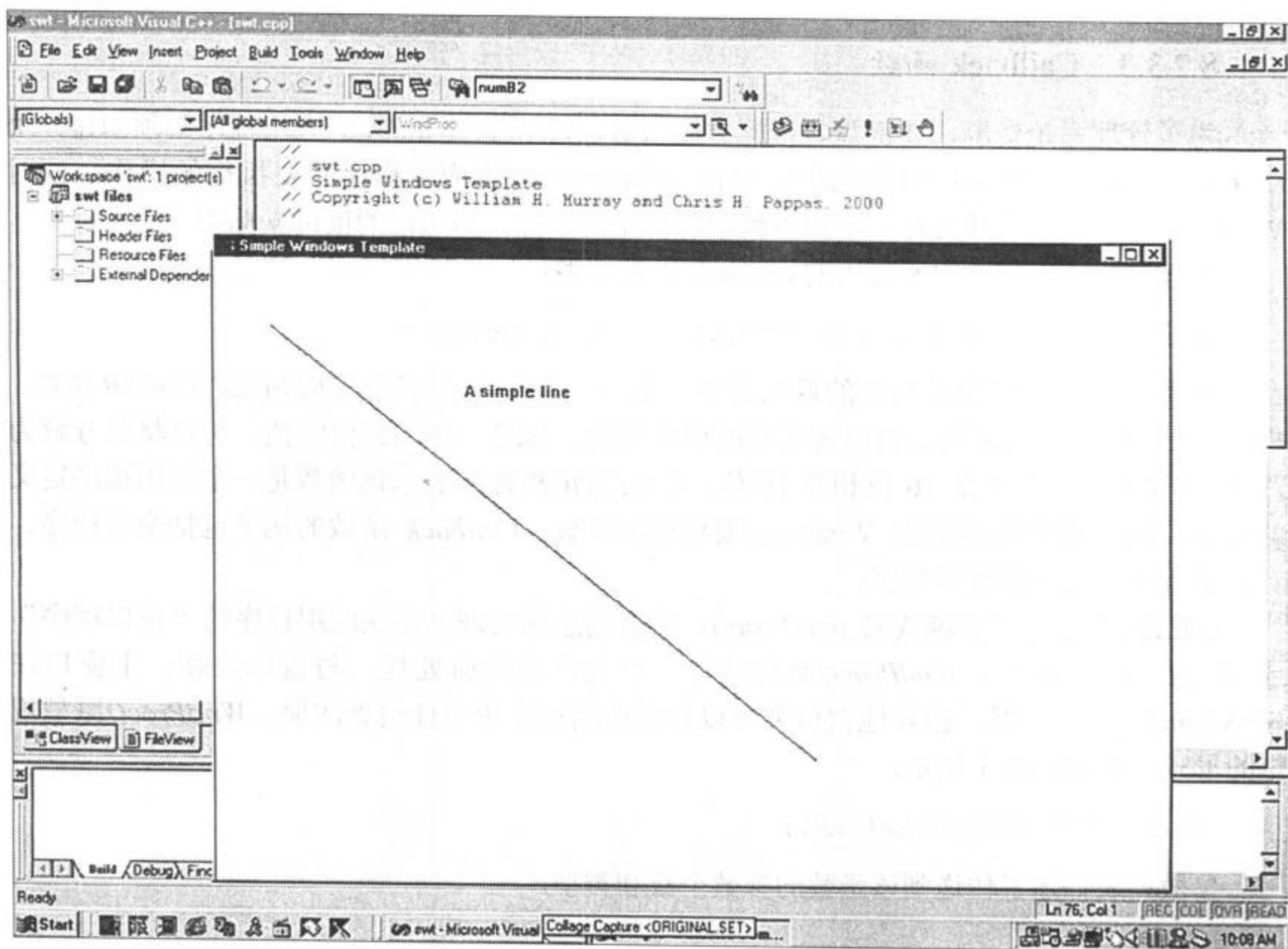


图 8-4 svt.cpp 程序执行时在屏幕上画出这一窗口

初始注释块的长度应该根据需要而定，以通知或提示用户关于该应用程序的所有具体情况。

8.2.3.2 <windows.h>

源文件接下来是如下的语句：

```
#include <windows.h>
```

包含语句引入一个名为 `windows.h` 的文本文件，其中包含了 Windows 的特殊定义和其他一些 `#include` 语句。这些文件的内容提供了构造 Windows 应用程序所需的基本定义。在 Visual C++ 头文件集中找出 `windows.h` 头文件的位置，检查其中所包含的大量的 Windows 信息。



8.2.3.3 Callback 函数

为了管理系统资源，如键盘、鼠标、硬盘等，每一个 Windows 应用程序必须创建一个 callback 函数。Callback 函数利用消息向 Windows 报告应用程序需要执行的操作，同时 Windows 也使用消息通知每一个应用程序所发生的事件，如当前的鼠标坐标等。

下面是该应用程序中所使用的 callback 函数原型：

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
```

所有的函数原型均以函数的返回类型开始——此处的例子为 *LRESULT CALLBACK*。*LRESULT* 部分定义了用于窗口过程返回值的类型，这是一个 32 位的值，可以将其分解为两个 16 位的值，称为低 16 位和高 16 位。*CALLBACK* 部分表示该函数是一个应用程序定义的函数，系统或子系统(例如 Windows)要调用该函数，*Callback* 函数的例子包括窗口程序、对话框程序以及异常处理程序。

CALLBACK 之后是函数名 *WndProc()*，所有消息的处理都是在应用程序的“窗口程序”或 *WndProc()* 中执行。*WndProc()* 函数通过一个窗口类注册处理，与窗口关联。主窗口在 *WinMain()* 函数中注册，但其他窗口类可以在该应用程序中的任何处注册。*WndProc()* 原型规定的形式参数列表如下所示：

```
(HWND, UINT, WPARAM, LPARAM) ;
```

参数列表描述了传送到该函数的参数个数和类型。

说明

C/C++ 代码风格习惯上使用全部大写的标识符定义非标准 C/C++ 数据类型。为查看这些大写标识符的定义，需要找到通过 *windows.h* 所引用的定义它们的头文件。

HWND 参数表示一个窗口句柄。*UINT* 参数是一个可移植的无符号整数类型，其长度由主机环境决定(Windows NT 和 Windows 98 为 32 位)。*WPARAM* 参数是用于声明 *wParam* 的类型，*wParam* 是一个窗口程序的第三个参数(一种多态[polymorph]数据类型)。最后，*LPARAM* 参数是用于声明 *lParam* 的类型，*lParam* 是窗口程序的第四个参数。

8.2.3.4 句柄

编写一个面向过程的 Windows 应用程序总是要涉及到句柄的使用，句柄是一个用来标识许多不同类型的对象，如窗口、控件、菜单、图标、钢笔和刷子、内存分配、输出设备、以及窗口实例等的唯一数字。在 Windows 术语中，每一个加载的程序副本均称为一个实例。

由于 Windows 允许同时运行同一个应用程序的多个副本，所以需要保持跟踪每一个这样的实例，而做到这一点就是通过为每一个正在运行的该应用程序副本附加一个唯一的实例



句柄实现。

8.2.3.5 WinMain()函数

所有的应用程序都必须有一个最小化的 `callback` 函数和一个名为 `WinMain()` 的函数，`WinMain()` 函数是一般程序执行开始和结束的处。

```
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,
                  LPSTR lpszCmdLine, int nCmdShow)
```

在这一原型中，`WINAPI` 表示用于 Win32 API 的调用规范，并且使用在 API 声明中的 *FAR PASCAL* 位置。`WinMain()` 函数由 Windows 作为基于 Win32 应用程序的初始入口点调用。`WinMain()` 函数将注册应用程序的窗口类类型，执行任何必须的初始化，并创建和初始化该应用程序的消息处理循环。

第一个参数 `hInst` 包含了该应用程序的当前实例的实例句柄，它唯一地标识了运行于 Windows 下的该程序。第二个参数 `hPreInst` 在 Windows 95 和 NT 下将包含一个 `NULL` 值，它表示该应用程序不存在先前的实例。在 Windows 98 和 NT(2000) 下，每一个应用程序都运行在其自己独立的地址空间。此处，`hPreInst` 将永远不返回有效的先前实例，而只是 `NULL`。第三个参数 `lpszCmdLine` 是一个指向空字符结束的字符串的指针，该字符串表示的是应用程序的命令行参数。如果应用程序以 Windows 的 Run 命令启动，则该值将为 `NULL`。保存在第四个参数 `nCmdShow` 中的 `int` 值表示的是 Windows 事先定义的许多常量之一，这些常量定义了一个窗口可以显示的各种可能的方式，例如 `SW_SHOWNORMAL`、`SW_SHOWMAXIMIZED` 或 `SW_SHOWMINIMIZED`。

`WinMain()` 的目的是初始化应用程序，显示其主窗口，并进入消息提取和发送循环，这种循环是对应用程序的其余部分的执行的最高层控制。`WinMain()` 还负责在接收到 `WM_QUIT` 消息时结束消息循环。

WNDCLASS `WNDCLASS` 结构中包含调用 `RegisterClass()` 所使用的窗口类属性，该结构的一般定义形式如下：

```
typedef struct _WNDCLASS {
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HANDLE      hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCTSTR     lpszMenuName;
```



```
    LPCTSTR lpszClassName;  
} WNDCLASS;
```

让我们检查每一个结构成员的用途，*style* 成员定义了类风格，如 *CS_BYTEALIGNCLIENT*、*CS_HREDRAW*、*CS_VREDRAW* 等。*lpfnWndProc* 成员是一个指向窗口程序的指针，*cbClsExtra* 成员指定分配在窗口类结构后面的额外字节数，*cbWndExtra* 成员指定分配在窗口实例之后的额外字节数，*hInstance* 成员是该窗口过程所在的实例句柄，*hIcon* 成员是类图标句柄，*hCursor* 成员是类光标的句柄，*hbrBackground* 成员是类背景刷子的句柄。*lpszMenuName* 成员是一个指向空字符结束的字符串指针，它指定了类菜单的资源名字，正如在资源文件中出现的那样。*lpszClassName* 成员是一个空字符结束的字符串的指针或原子。如果该参数是一个字符串，则指定窗口类的名字。

RegisterClass() *WinMain()* 函数中的下一条语句是调用 **RegisterClass()** 函数：

```
if (!RegisterClass (&wcApp))  
    return 0;
```

为 Windows 应用程序所创建的每一个窗口都必须基于一个窗口类。*WinMain()* 注册应用程序的主窗口类，每一个窗口类都基于用户选择的一组数据：风格、字体、标题栏、图标、长度、位置等。窗口类作为一个定义了这些属性的模板使用。

if 语句注册了一个新的窗口类，方法是给予 **RegisterClass()** 一个指向窗口类结构的指针。如果 Windows 没有注册该窗口类，可能是由于缺少内存，则 **RegisterClass()** 返回一个 0，然后结束该程序。

在 Windows 98 或 NT 下，**RegisterClassEx()** 函数可以用来代替 **RegisterClass()** 函数，**RegisterClassEx()** 函数允许包含小图标。

CreateWindow() 窗口的建立通过调用 Windows 的 **CreateWindow()** 函数完成。当窗口类定义了一个窗口的一般特性时，则允许同一窗口类在许多不同的窗口中使用，**CreateWindow()** 的参数指定了更多关于该窗口的详细信息。

CreateWindow() 函数使用传送给它的信息描述了该窗口的类、标题、风格、屏幕位置、父句柄、菜单句柄以及实例句柄。为模板应用程序调用 **CreateWindow()** 使用如下的实际参数：

```
hWnd=CreateWindow(szProgName, "Simple Windows Template",  
                  WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,  
                  CW_USEDEFAULT, CW_USEDEFAULT,  
                  CW_USEDEFAULT, (HWND) NULL, (HMENU) NULL,  
                  (HANDLE) hInst, (LPSTR) NULL);
```

第一个字段 *szProgName* 定义了该窗口的类，其后是用于该窗口的标题。窗口的风格为第三个参数 (*WS_OVERLAPPEDWINDOW*)。这种标准的 Windows 风格表示了一个具有标题栏、系统菜单框、最小化和最大化图标以及窗口框架的普通重叠窗口。



后面的六个参数(无论是 *CW_USEDEFAULT* 还是 *NULL*)表示窗口的初始位置的 *x*、*y* 和窗口长度的 *x*、*y*，以及父窗口句柄和窗口菜单句柄。每一个这样的字段都指定了一个缺省值。*hInst* 字段包含了该程序的实例句柄，后面是一个表示无附加参数的值(*NULL*)。

如果成功，则 *CreateWindow()* 返回新创建窗口的句柄，否则，该函数返回 *NULL*。

显示和更新窗口 为显示一个窗口，需要调用 Windows 的 *ShowWindow()* 函数。

```
ShowWindow(hWnd, nCmdShow);
```

hWnd 参数保存的是窗口的句柄，而 *nCmdShow* 的值指定了该窗口将以普通窗口 (*SW_SHOWNORMAL*)，或其他几种可能方式显示。

显示一个窗口的最后一步需要调用 Windows 的 *UpdateWindow()* 函数。

```
UpdateWindow(hWnd);
```

正是对 *UpdateWindow()* 的这一调用，引起客户区域通过产生 *WM_PAINT* 消息而得到绘制。关于 *WM_PAINT* 的说明在本章稍后给出。

消息循环 Windows 并不直接发送鼠标或键盘输入信息给一个应用程序，而是将所有的输入放置到该应用程序的消息队列中。消息队列中可以包含由 Windows 产生的消息或由其他应用程序传送来的消息。

一旦对 *WinMain()* 的调用开始维护创建和显示该窗口，应用程序就需要创建一个消息处理循环，最常用的方法是使用标准 C++ 的 **while** 循环。

```
while (GetMessage(&lpMsg, NULL, 0, 0))
{
    TranslateMessage(&lpMsg);
    DispatchMessage(&lpMsg);
}
```

一个对 *GetMessage()* 函数的调用从应用程序的消息队列中提取下一条将要处理的消息。*GetMessage()* 将消息复制到由长型指针 *lpMsg* 指向的消息结构中，并将其发送到该程序的主体中。

DispatchMessage() 函数将虚拟键消息转化为字符消息，只有需要处理来自键盘的字符输入的应用程序才需要调用该函数。*DispatchMessage()* 函数用于将当前消息发送到正确的窗口过程。

WinMain() 一般来说 *WinMain()* 负责当接收到一个 *WM_QUIT* 消息时结束消息循环，这将结束该应用程序，然后返回在 *WM_QUIT* 消息的 *wParam* 参数中所传送的值。

8.2.3.6 所需的窗口函数

所有的 Windows 应用程序都必须包含一个 *WinMain()* 和一个 Windows 回调函数。由于



Windows 应用程序从来不直接访问任何 Windows 函数，所以每一个应用程序都必须给出一个对 Windows 的请求，以执行特定操作。

说明

切记，回调函数由 Windows 注册，并且当 Windows 在一个窗口上执行一个操作时回调。对每一个应用程序来说，回调函数的实际大小不同。窗口函数本身可能很小，仅仅处理一条或两条消息，它也可能很大很复杂。

Windows 中有数百条可以发送到窗口函数的不同的消息。所有这些消息都以“WM_”开始的标识符标记。在 swt.cpp 程序中，只涉及到了 WM_PAINT 和 WM_DESTROY 两条消息。

WndProc()函数通过定义若干个变量继续，在该例中 *hdc* 指的是显示上下文句柄，*ps* 指的是需要用来保存客户区域信息的 PAINTSTRUCT 结构。回调函数的主要目的是检查将要处理的消息的类型，然后选择所要采取的适当操作。这一选择处理通常采用标准 C++ 开关语句的形式。

WM_PAINT 消息 在这一程序中，WndProc()将要处理的第一条消息是 WM_PAINT。该消息调用 Windows 的函数 BeginPaint()，为绘制准备一个指定窗口(*hWnd*)，并使用关于绘制区域的信息填充一个 PAINTSTRUCT(&*ps*)。BeginPaint()函数还返回一个关于给定窗口的设备描述表句柄。

设备描述表使用缺省的笔、刷子及字体设置。缺省的笔为黑色、一个像素宽、绘制实线。缺省的刷子是一个具有实心形式的白色刷子。缺省的字体为系统字体。这种设备描述表非常重要，其原因是 Windows 应用程序所使用的所有显示函数都需要一个指向设备描述表的句柄。

通过调用 InvalidateRect()函数，应用程序可以强制一条 WM_PAINT 消息，InvalidateRect()函数标记该应用程序的客户区域为无效的。通过调用 GetUpdateRect()函数，应用程序可以获得无效矩形的坐标。随后对 ValidateRect()函数的调用使客户区域的任何矩形区域有效化，并删除所有挂起的 WM_PAINT 消息。

WndProc()函数通过调用 EndPaint()函数，结束其对 WM_PAINT 消息的处理。当应用程序完成输出信息到客户区域时，则调用 EndPaint()函数。该函数通知 Windows：应用程序已经完成了对所有绘制消息的处理，现在已为删除显示的上下文做好准备。

WM_DESTROY 消息 当用户从系统菜单中选择了 Close 选项时，Windows 传递一个 WM_DESTROY 消息到应用程序的消息队列中，当提取到该消息后程序结束。

DefWindowProc()函数 DefWindowProc()函数的调用在 WndProc()函数的开关语句的 default 部分进行，起作用清除应用程序的消息队列中不可识别——当然也就是未处理的——消息。该函数保证了所有传送到应用程序的消息都得到处理。



8.3 调试

本节中我们将演示一些在创建应用程序时可能遇到的面向过程的 Windows 程序错误，然后我们将讲解如何使用 Debugger 跟踪和纠正这些问题。这些问题可以分为两个大类，可能遇到的第一类问题处理边界问题。边界问题通常是由于错误使用屏幕长度所造成的，包括不正确地使用窗口的范围、视口的范围、客户区域等；第二类问题包括对添加到程序的资源的使用。在此处的例子中我们演示了一个粗心的程序员，在不知道 Windows 如何分配某些资源参数的情况下，是如何陷入困境的。

8.3.1 一个动画位图程序

通过使用一个简单的绘画、擦除、移动、再绘画机制，可以使用位图图像建立动画效果。对于此处的例子，位图图像绘制在屏幕上，并且允许停留在那里一段时间，然后擦除该图像。擦除图像可以使用三种技术完成：以背景颜色重画图像，清除一个环绕该图像的矩形区域，或清除整个屏幕。下一个图像的坐标将向左、右、上、下、或者这些方向中两个方向的结合方向移动若干个像素，然后重新绘制该图像。当时间循环恰当时，观察者在观看窗口时将产生一种移动的感觉。

在这一程序中，`BitBlt()`函数用来在屏幕上移动一个飞碟的位图图像。图 8-5 给出了在 Resource Editor 中设计该位图时的图像。

`saucer.rc` 资源脚本文件用于标识针对该工程的位图资源和独特的图标，下面是该资源脚本文件的一个简化部分：

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

.
.
.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Bitmap
//
```

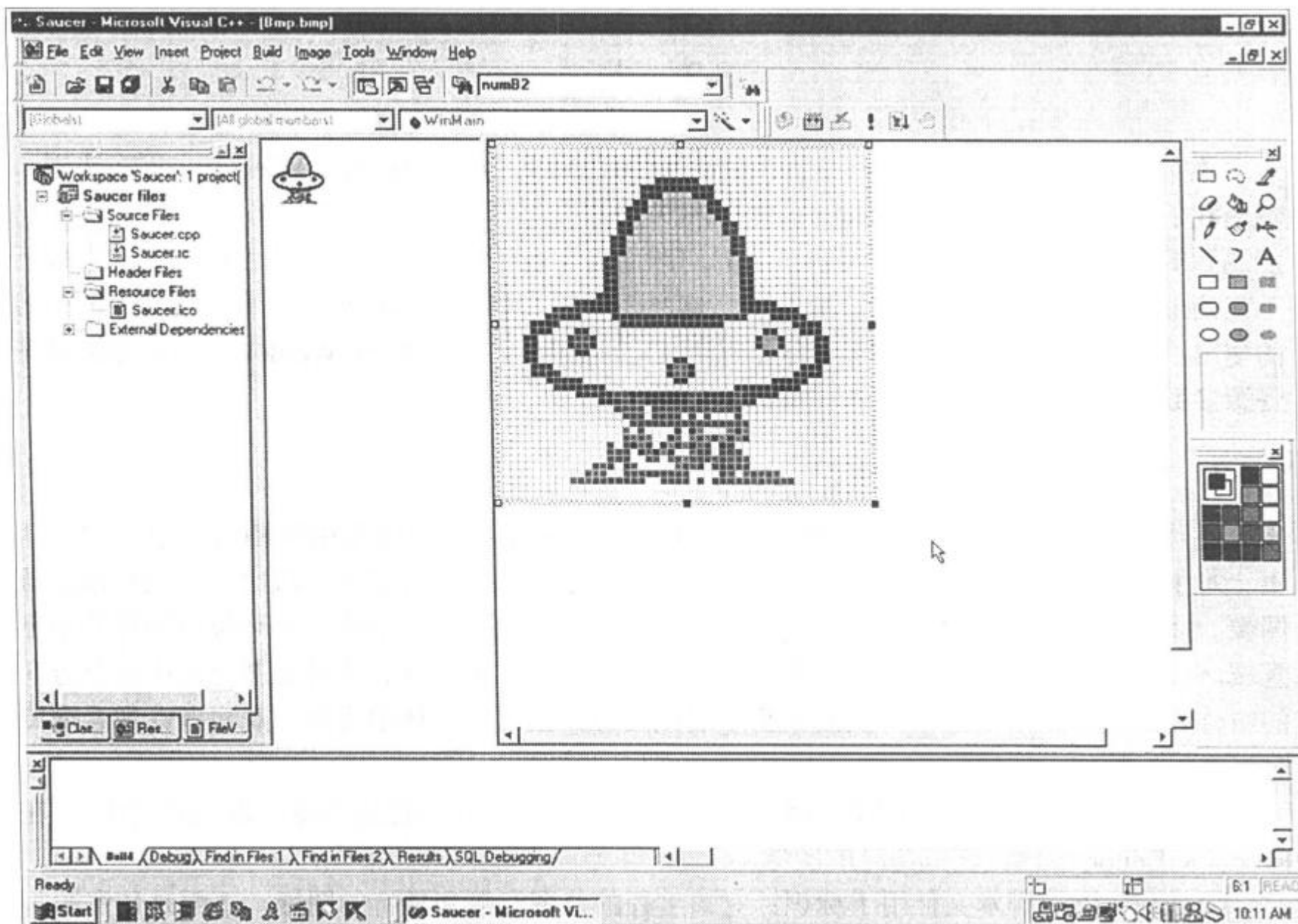


图 8-5 bmp.bmp 是飞碟的位图图像

```
BMIMAGE BITMAP DISCARDABLE "Bmp.bmp"
#ifdef APSTUDIO_INVOKED
////////////////////////////////////
.
.
.
////////////////////////////////////
//
// Icon
// Icon with lowest ID value placed first to ensure application
// icon Remains consistent on all systems.
SAUCERICON ICON DISCARDABLE "Saucer.ico"
#endif //English (U.S.) resources
////////////////////////////////////
.
.
.
```



资源编辑器产生的文件为 saucer.rc，而在建立可执行程序时资源编译器所创建的文件是 saucer.res。要在这些文件中找出错误几乎是不可能的。

源代码 saucer.cpp 建立在前一节的 swt.cpp 模板之上，检查如下的代码：

```
//
// A program with problems
// A procedure-oriented Windows Application
// that demonstrates simple animation techniques
// with a bitmapped images.
// Copyright (c) William H. Murray and Chris H. Pappas, 2000
//
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
char szProgName[]="ProgName";
char szIconName[]="SaucerIcon";
char szBMName[]="BMImage";
HBITMAP hBitmap;
int iTimer,xPos,yPos;
int xPosInit,yPosInit,xStep,yStep;
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE hPreInst,
                  LPSTR lpszCmdLine,int nCmdShow)
{
    HWND hWnd;
    MSG lpMsg;
    WNDCLASS wcApp;

    wcApp.lpszClassName=szProgName;
    wcApp.hInstance    =hInst;
    wcApp.lpfnWndProc  =WndProc;
    wcApp.hCursor      =LoadCursor(NULL, IDC_ARROW);
    wcApp.hIcon        =LoadIcon(hInst, szIconName);
    wcApp.lpszMenuName =0;
    wcApp.hbrBackground=(HBRUSH) GetStockObject(WHITE_BRUSH);
    wcApp.style        =CS_HREDRAW|CS_VREDRAW;
    wcApp.cbClsExtra   =0;
    wcApp.cbWndExtra   =0;
    if (!RegisterClass (&wcApp))
        return 0;

    hWnd=CreateWindow(szProgName,"Flying Saucer Program",
                     WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,
                     CW_USEDEFAULT,CW_USEDEFAULT,
                     CW_USEDEFAULT, (HWND)NULL, (HMENU) NULL,
                     hInst, (LPSTR) NULL);

    ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);
}
```



```
// load saucer bitmap
hBitmap=LoadBitmap(hInst,szBMName);
while (GetMessage(&lpMsg,0,0,0)) {
    TranslateMessage(&lpMsg);
    DispatchMessage(&lpMsg);
}
return(lpMsg.wParam);
}
LRESULT CALLBACK WndProc(HWND hWnd,UINT messg,
                        WPARAM wParam,LPARAM lParam)
{
    HDC hdc;
    HDC hmdc;
    static BITMAP bm;
    switch (messg)
    {
        case WM_CREATE:
            // initial values
            xPosInit=200;
            yPosInit=200;
            xStep=4;
            yStep=4;
            xPos=xPosInit;
            yPos=yPosInit;
            iTimer=SetTimer(hWnd,1,10,NULL);
            break;
        case WM_TIMER:
            // with each timer tick, draw image
            hdc=GetDC(hWnd);
            hmdc=CreateCompatibleDC(hdc);
            xPos+=xStep;
            yPos+=yStep;
            // draw image
            SelectObject(hmdc,hBitmap);
            GetObject(hBitmap,sizeof(bm),(LPSTR) &bm);
            BitBlt(hdc,xPos,yPos,bm.bmWidth,bm.bmHeight,
                hmdc,0,0,SRCCOPY);
            // check left and right window edges
            if((xPos > 639) ||
                (xPos < 0)) {
                xStep=-xStep;
            }
            // check top and bottom window edges
            if((yPos > 479) ||
                (yPos < 0)) {
                yStep=-yStep;
            }
    }
}
```



```

    }
    ReleaseDC(hWnd, hdc);
    DeleteDC(hmdc);
    break;
case WM_DESTROY:
    if (hBitmap) DeleteObject(hBitmap);
    if(iTimer) KillTimer(hWnd, 1);
    PostQuitMessage(0);
    break;
default:
    return(DefWindowProc(hWnd, messq, wParam, lParam));
    break;
}
return(0);
}

```

检查 saucer.cpp 源代码并注意在两个方向上所做的移动，飞碟可能跳出窗口的顶部、底部、右边或左边。随着每一个所处理的计时器消息(WM_TIMER)，程序将绘制出一个位图图像。

```

//draw image
SelectObject(hmdc, hBitmap);
GetObject(hBitmap, sizeof(bm), (LPSTR) &bm);
BitBlt(hdc, xPos, yPos, bm.bmWidth, bm.bmHeight,
        hmdc, 0, 0, SRCCOPY);

```

飞碟动画从两个方向上获得，下面是负责在屏幕上移动飞碟的代码：

```

//check left and right window edges
if((xPos > 639) ||
    (xPos < 0)) {
    xStep=-xStep;
}
//check top and botttom window edges
if((yPos > 479) ||
    (yPos < 0)) {
    yStep=-yStep;
}

```

一条 if 语句控制了左右方向的移动，另一条 if 语句则控制了上下方向的移动。对于每一个时钟滴答，均将指定一个新的屏幕位置。这两条语句检查水平和垂直方向上懂得移动值，以保证新的屏幕位置在窗口边界范围内。如果新位置在窗口范围内，则一切正常。如果新位置不在窗口范围内，则说明在位图图像与边界之间发生了冲突。此时，改变 xStep 或 yStep 增量的符号，使其朝相反方向移动。这至少是所设想的工作方式。



8.3.1.1 含有问题的代码

如果编译并运行该程序，可以发现该应用程序存在一些问题，如图 8-6 所示。

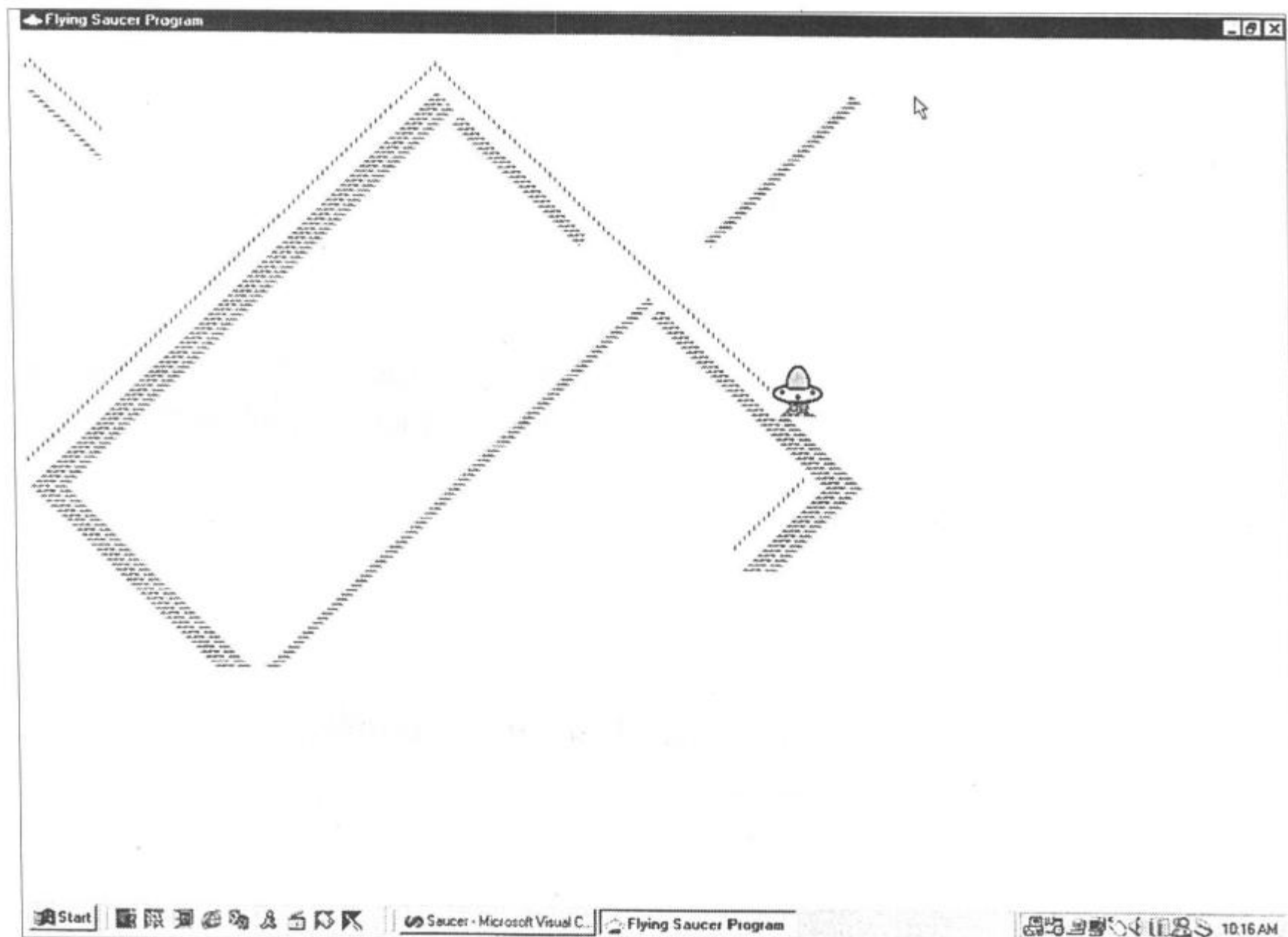


图 8-6 飞碟程序似乎存在一些问题

首先，当飞碟在两个方向上移动时，在这些方向上似乎并没有受到屏幕边界的控制，为什么出现这种情况？其次，飞碟在屏幕上移动时留下了一条轨迹，造成这种情况的原因是什么？最后当图像与窗口的右边界和底边界接触时，图像的大部分在其向相反方向移动之前就已消失。让我们使用 Debugger 帮助跟踪这些问题。

使用一台计算机调试会话所存在的问题 这是说明为什么需要两台网络连接的计算机，以可以利用在上一章中所描述的远程调试特性的绝佳时机。

为了说明这一问题，编译该程序并启动 Debugger。在如下所示的代码段中黑体显示的行上设置一个断点。



```
//check top and bottom window edges
if((yPos > 479) ||
    (yPos < 0)) {
    yStep=-yStep;
}
ReleaseDC(hWnd,hdc);
DeleteDC(hmdc);
break;
```

现在，使用 Go(F5)命令运行到该断点。应用程序启动。为观察飞碟窗口，只要使用 ESC+ALT 组合键切换活动窗口即可。发生了什么事情？图 8-7 说明了一切。

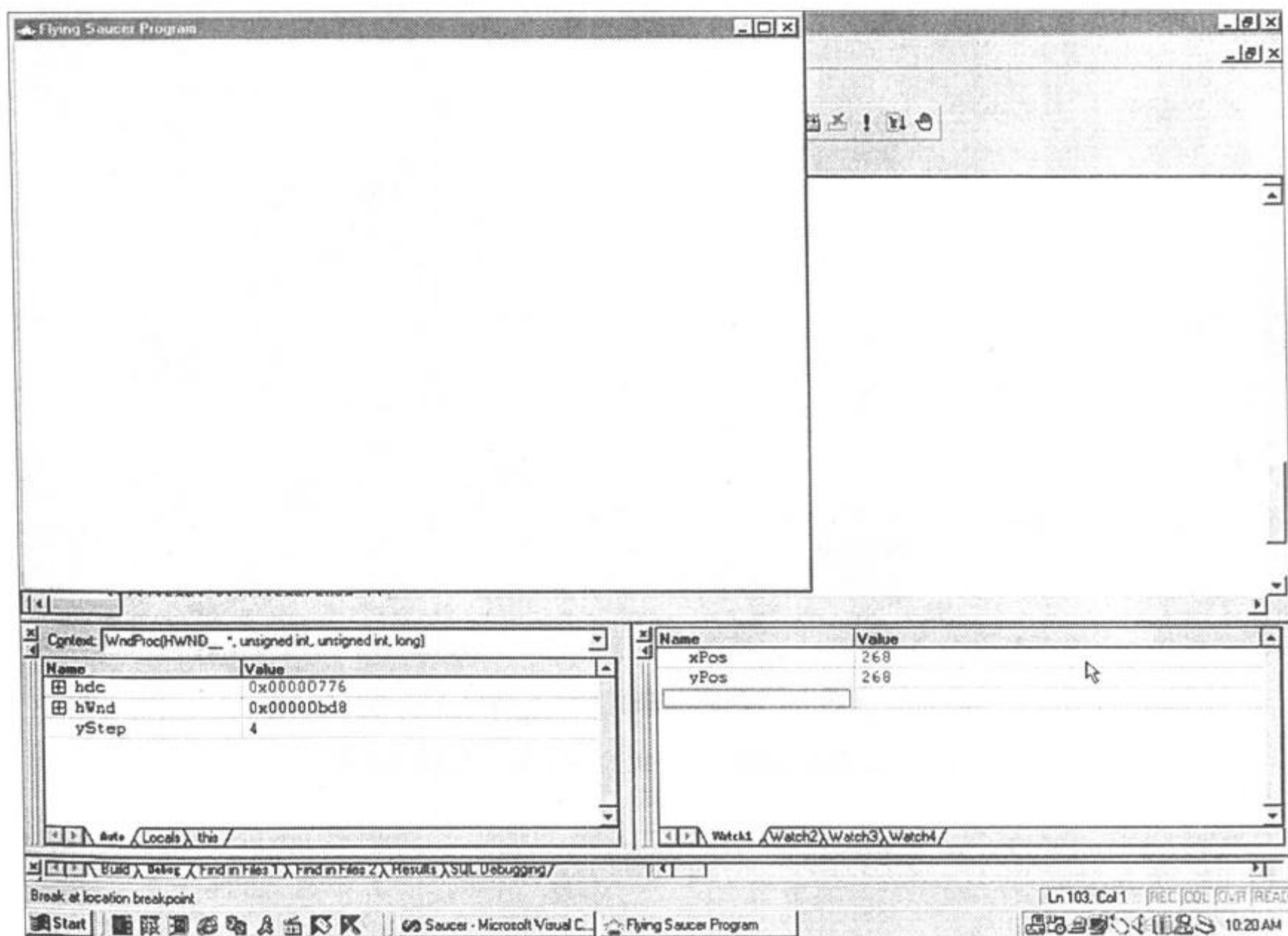


图 8-7 当飞碟窗口移动到前台时，位图图像不可见

这是一个很大的窗口，但 Debugger 并没有与应用程序共享这一窗口，解决的方法是使用远程调试。



按照本章前面所给出的步骤启动一个远程调试会话。回想一下，当时的远程目标计算机是一台 Sony 机(网络上标识为“sony”)，主计算机为一台 Hewlett-Packard 机(标识为“hp”)。图 8-8 给出了与前面相同的调试位置，但此处是在一个使用远程目标计算机的系统上。

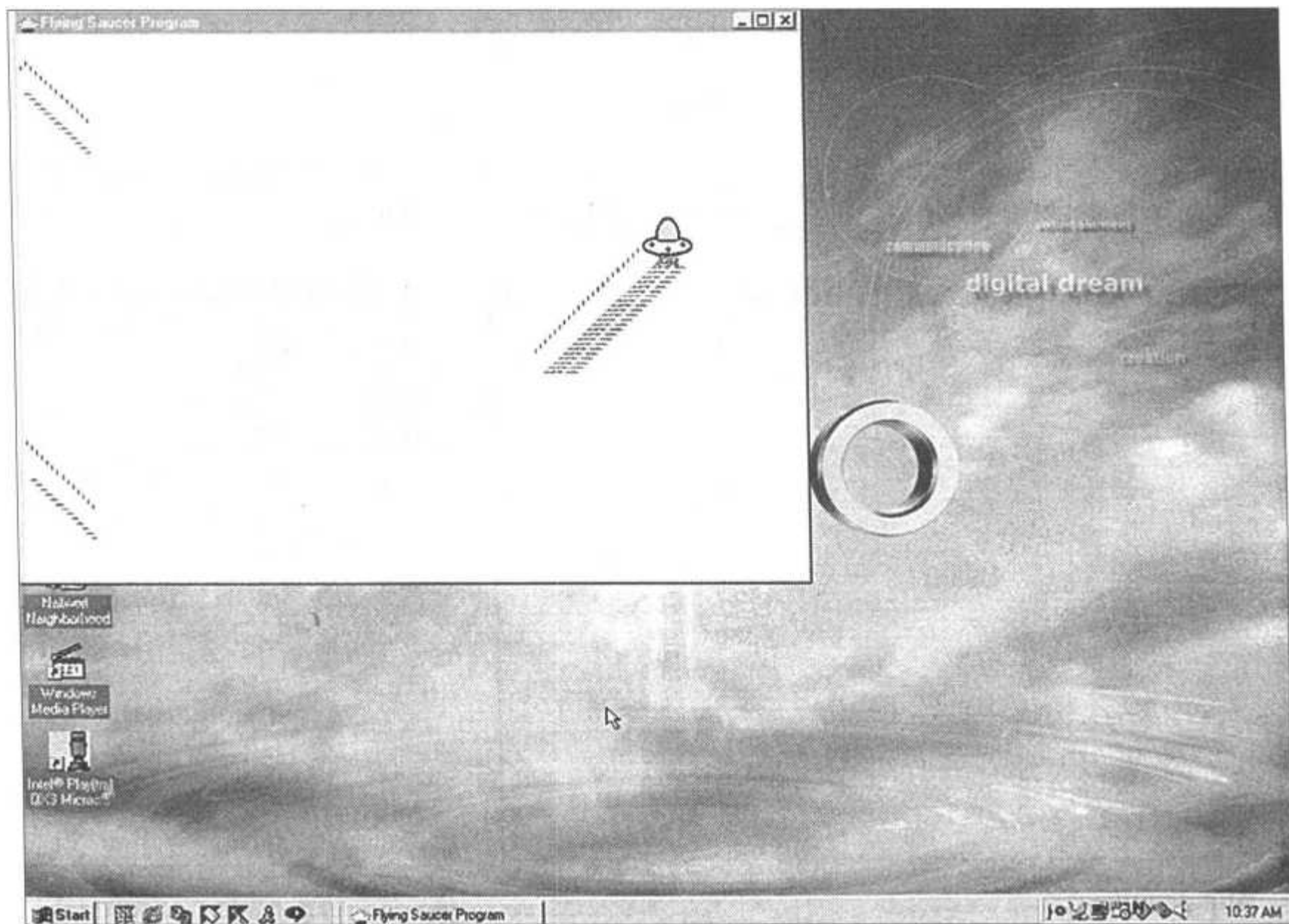


图 8-8 飞碟位图在远程目标计算机(sony)上是可见的

同时主计算机(hp)上可以看到调试位置和调试信息，如图 8-9 所示。

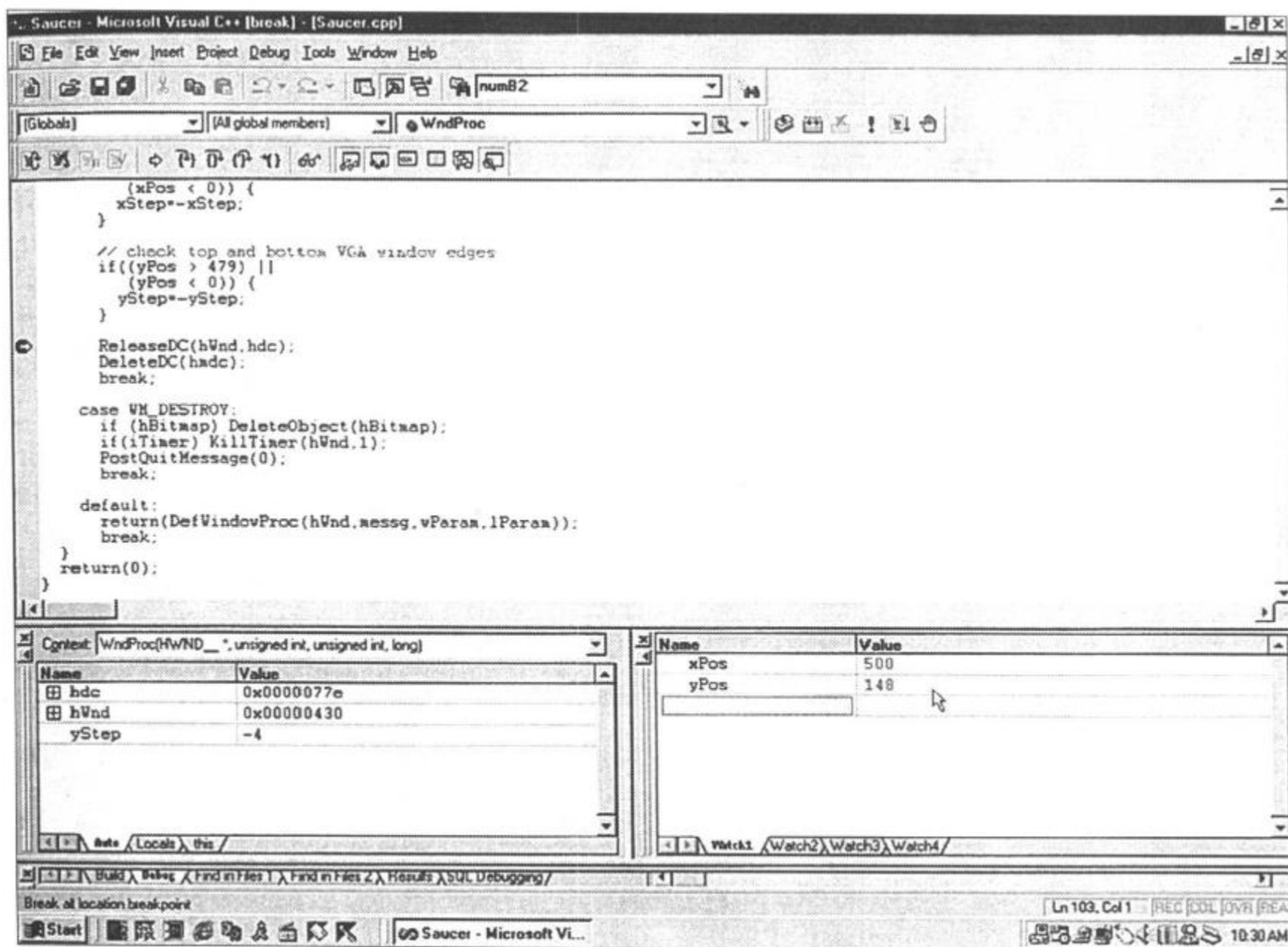


图 8-9 在主计算机(hp)上可同步看到调试信息

下一部分中我们将开始跟踪 Flying Saucer 程序中所存在的问题。

使用远程目标计算机调试 我们要跟踪的第一个问题是，为什么飞碟图像侵入到窗口的右边界和底边界，而在窗口的左边界和上边界处则运行正常。

图 8-10 给出了图像跌落到窗口底边界之下的情况。

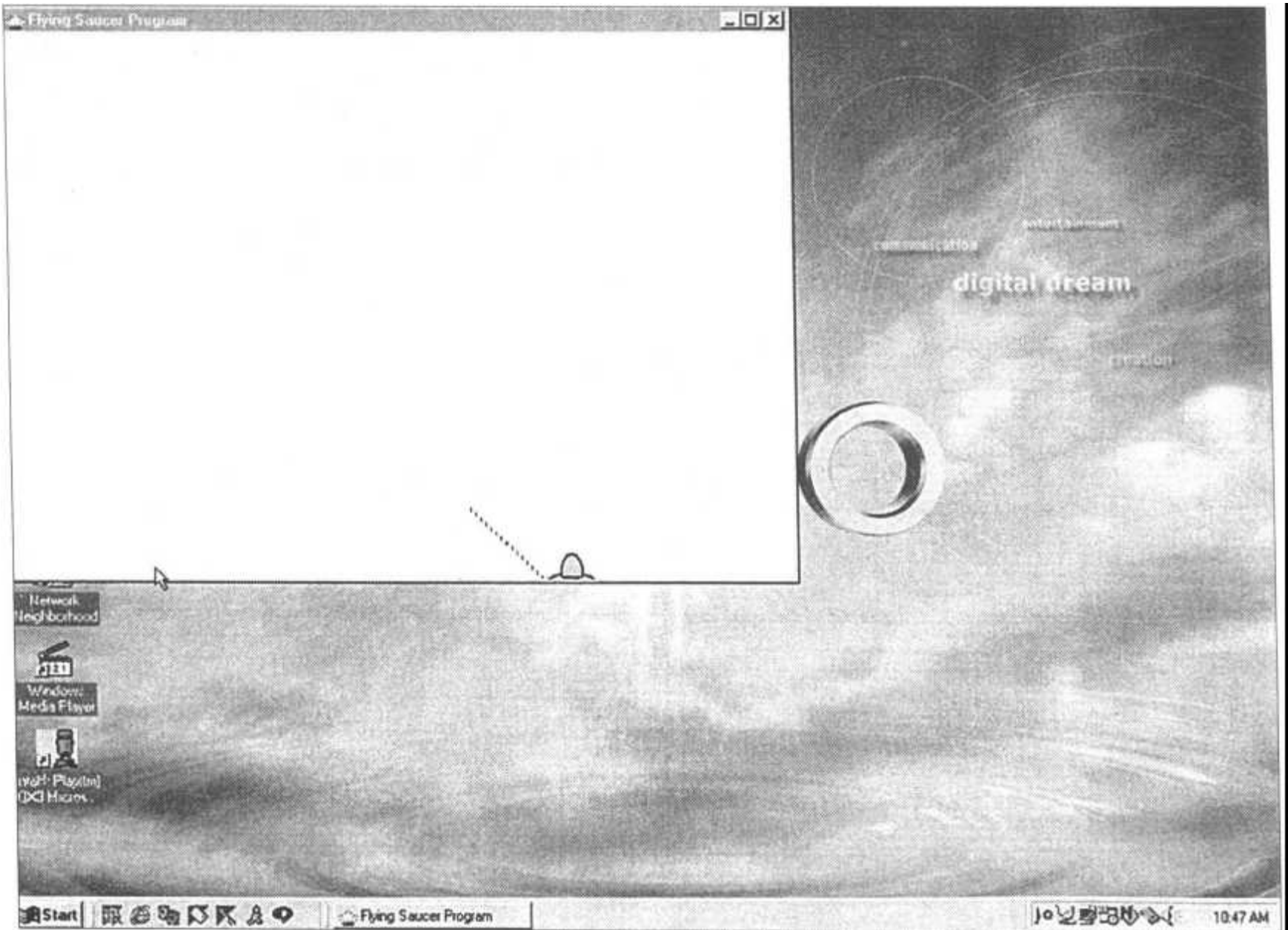


图 8-10 飞碟侵入了窗口底边界而不是从该边界反弹回去(远程目标计算机——sony)

启动调试会话，将断点置于与前一节中完全相同的位置。添加 *xPos* 和 *yPos* 变量到观察窗口。现在，执行若干 Go(F5)选项，直至图像与窗口的右边界或底边界正好接触。让我们来看看其中的一种情况，如图 8-11 所示。

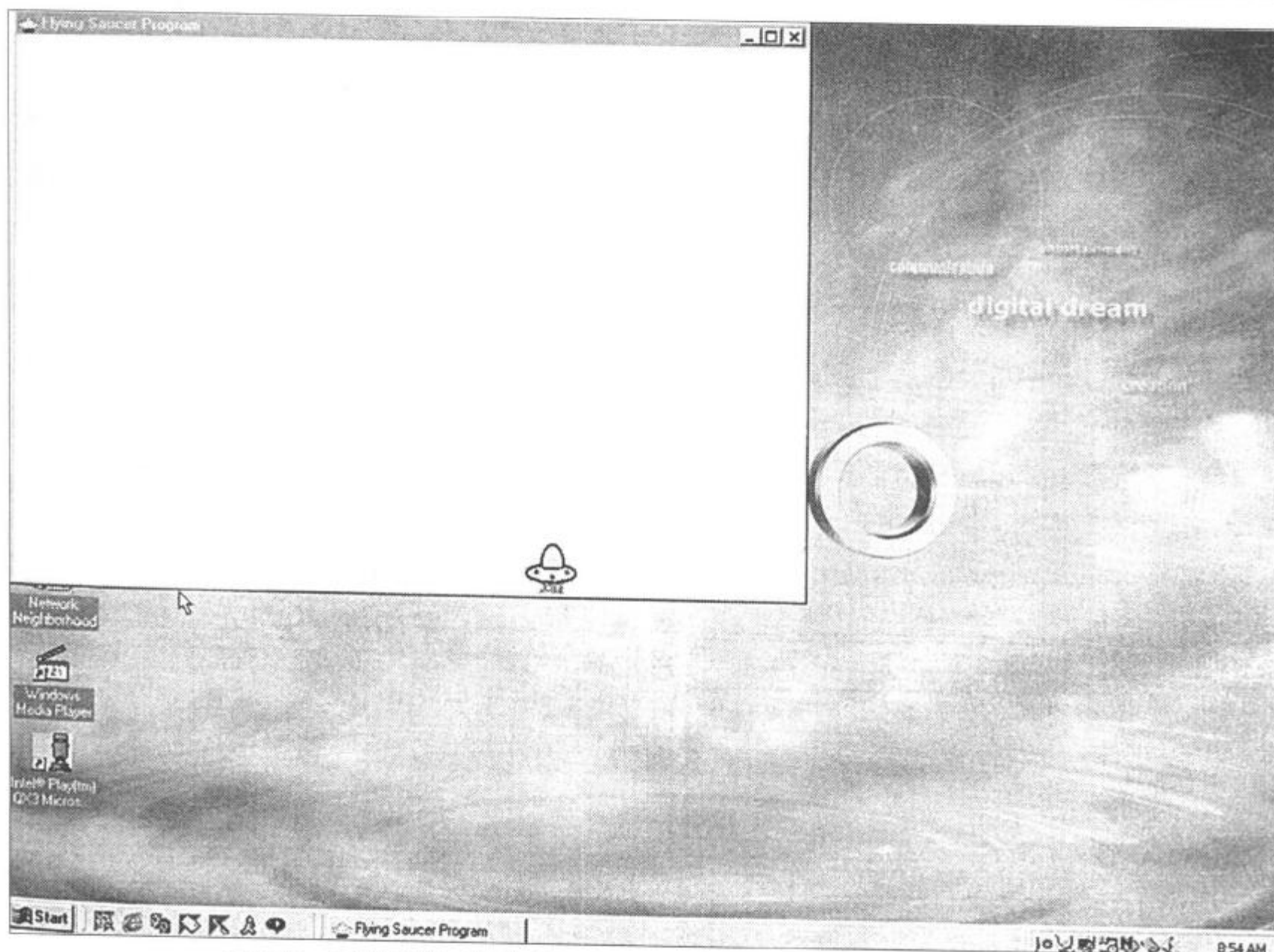


图 8-11 飞碟图像正好与窗口的底边界接触(远程目标计算机——sony)

现在检查观察窗口中的变量，如图 8-12 所示。

可以清楚地看到，通过检查 *yPos* 变量，图像并不如所希望的那样处于 479 像素的位置，而是在 408 位置。现在，移动飞碟使其正好与右边界接触。观察 *xPos* 变量并注意同样的问题：当飞碟与右边界碰撞时，*xPos* 变量的值为 588。按照如下清单中黑体所示修改边界值可以修复这两个问题：

```
//check left and right window edges
if ((xPos > 588) ||
    (xPos < 0)) {
    xStep=-xStep;
}
//check top and bottom window edges
if((yPos > 408) ||
    (yPos < 0)) {
    yStep=-yStep;
}
```

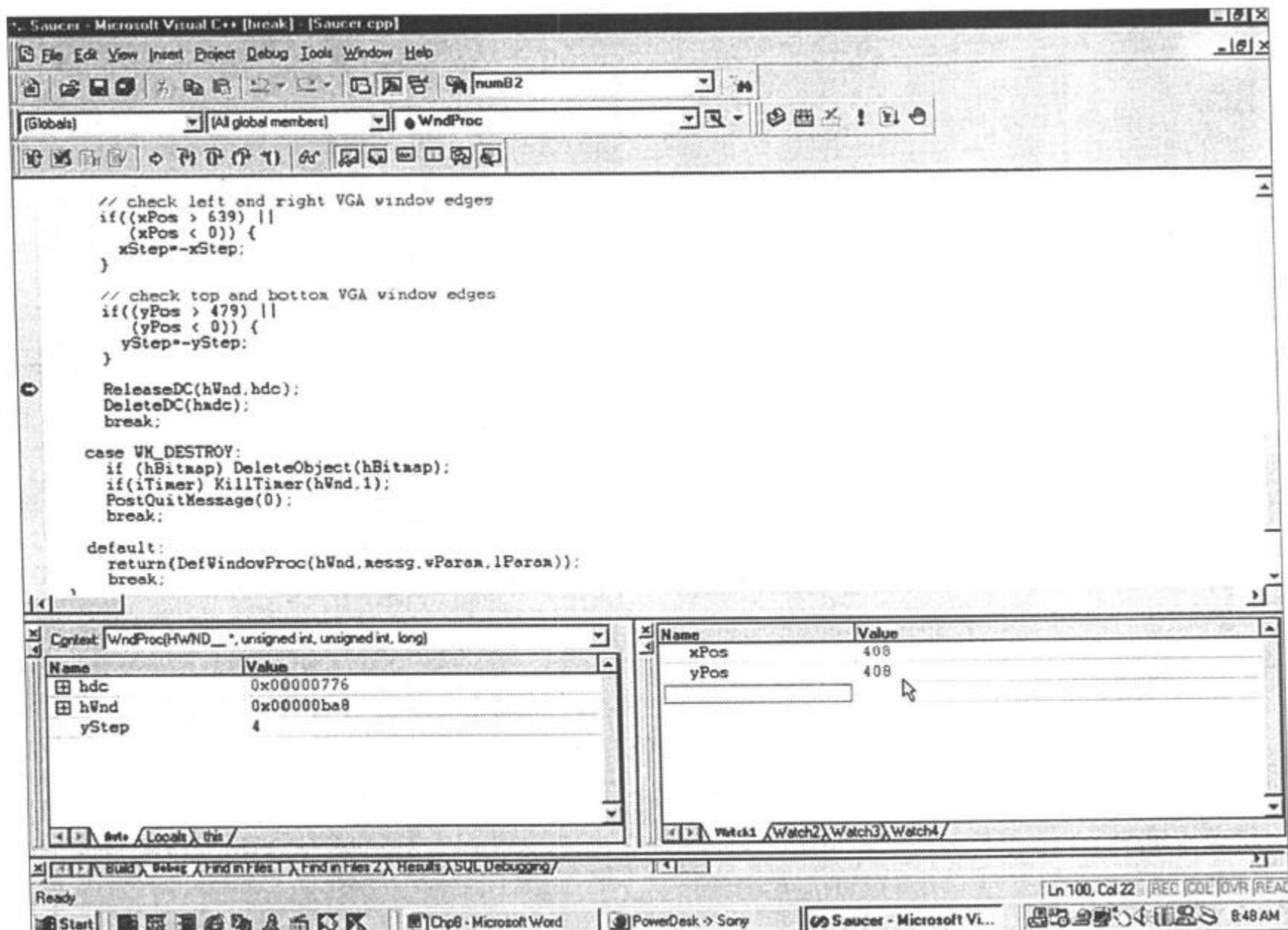


图 8-12 当飞碟与窗口的底边界接触时检查变量 xPos 和 yPos

我们已经找出了问题，但为什么原来的值不起作用？部分原因是位图的焦点位于其左上角，所以为了在图像与窗口的右边界或底边界碰撞时不丢失图像，必须从底边界和右边界值中减去位图的大小。在进入 `GetObject()` 函数时已经得到了位图的长度，位图的长度在 `bm` 结构中返回。要获得位图的宽度和高度，查看结构成员 `bm.bmWidth` 和 `bm.bmHeight` 即可。所以从理论上说，应该可以执行如下的操作，而无须根据位图大小采用“硬布线”方式处理边界的值：

```
//check left and right window edges
if ((xPos+ bm.bmWidth > 639) ||
    (xPos < 0)) {
    xStep=-xStep;
}
//check top and bottom window edges
if ((yPos+ bm.bmHeight > 408) ||
```



```
(yPos < 0)) {
    yStep=-yStep;
}
```

如果使用这些值运行程序，则可以看到程序运行要更好一些但仍不完美。问题仍然是与窗口的宽度与高度有关。在考虑窗口的客户区域时，我们错误地认为其宽度为 640 个像素，高度为 480 个像素。实际上，要比这短一些，其中还包含了窗口边界的宽度和标题栏宽度。我们可以通过不断减小像素直至恰好的位置进行试验，也可以试着确定客户区域的实际宽度和高度。

错误监视

作为程序员，更倾向于以屏幕分辨率考虑问题，如 640×480 或 1024×768。位于一个窗口内部的客户区域总是要比这些值短，即使是最大化的窗口。在计算客户区域的宽度和高度时，一定要考虑边界宽度和标题栏宽度。

调用 `GetClientRect()` 函数可以获得客户区域的宽度和高度，通过该函数所获得的信息返回到一个名为 `rcWnd` 的 `RECT` 结构中。使用这一信息，底边界和右边界处所存在的问题将永久性得到纠正。

```
//check left and right window edges
if ((xPos+ bm.bmWidth > rcWnd.right) ||
    (xPos < rcWnd.left)) {
    xStep=-xStep;
}
//check top and bottom window edges
if ((yPos+ bm.bmHeight > rcWnd.bottom) ||
    (yPos < rcWnd.top)) {
    yStep=-yStep;
}
```

这一解决方法还有一个附加的好处，即解决了前面所提到的另一个问题。在图 8-6 中我们看到，在屏幕的中央附近，飞碟似乎已经弹出了一个不可见的边界。产生这种现象的原因是我们将屏幕的宽度和高度设置为了 VGA(640×480) 模式，而图 8-6 所显示的窗口为 1024×768。图 8-6 中的客户区域要更大一些，而边界值仍未改动。上面的解决方案也同时解决了这一问题。

另一个问题是为什么飞碟会在屏幕上留下一条轨迹？看上去似乎是每一次移动是没有能正确地擦除图像，确实如此。再次检查一下图 8-5，注意在所有的边上该图像都有一个至少 2 个像素的边界与窗口的背景色相匹配。

对于该程序中所使用的技术，每次飞碟移动时，新图像正好绘制在原图像上。返回调试会话，并将变量 `xPos`、`yPos`、`xStep` 和 `yStep` 放入观察窗口，如图 8-13 所示。

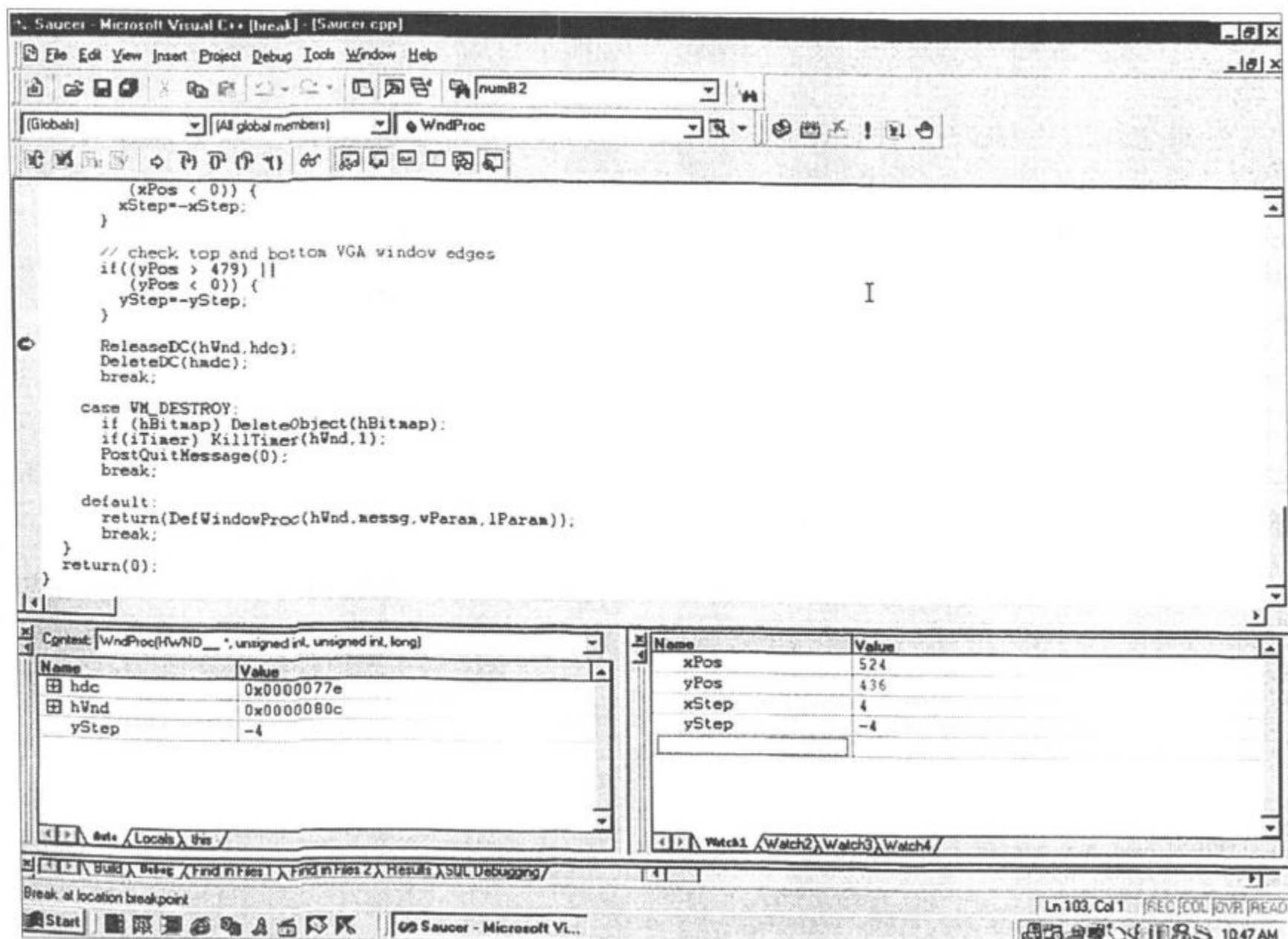


图 8-13 在观察窗口中检查四个变量

当我们进一步检查位图图像和每一个图像的步长大小，问题已经很清楚。如果步长大小为 4，但在飞碟周围的白色边界为 2，这样在每次重画时飞碟的一部分将保留在屏幕上。一种解决的方案是，绘制原始图像并停留片刻，然后画一个空图像(擦除屏幕上的矩形区域)，最后再在新的位置重新绘制图像。然而，有时候这种技术会在屏幕上产生闪烁，特别是对于速度较慢的计算机。另一种解决方案是确保步长大小总是等于或小于位图图像周围的边界宽度。这种方法确实是一个好方法，1 个像素的位图图像边界与 1 个像素的步长大小将使屏幕上的移动达到最平滑。

下一节中将看到对原始代码所做的所有修改。

8.3.1.2 好的代码

Debugger 已经帮助我们跟踪并纠正了 Flying Saucer 程序中的若干个问题。源代码清单



sarcer.cpp 中提供了所有这些问题。下面的清单中对源代码所做的所有修改以黑体字符表示：

```
//
// Saucer.cpp
// A procedure-oriented Windows Application
// that demonstrates simple animation techniques
// with a bitmapped images.
// Copyright (c) William H. Murray and Chris H. Pappas, 2000
//
#include <windows.h>
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
char szProgName[]="ProgName";
char szIconName[]="SaucerIcon";
char szBMName[]="BMImage";
HBITMAP hBitmap;
int iTimer,xPos,yPos;
int xPosInit,yPosInit,xStep,yStep;
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE hPreInst,
                    LPSTR lpzCmdLine,int nCmdShow)
{
    HWND hWnd;
    MSG lpMsg;
    WNDCLASS wcApp;

    wcApp.lpszClassName=szProgName;
    wcApp.hInstance    =hInst;
    wcApp.lpfnWndProc  =WndProc;
    wcApp.hCursor      =LoadCursor(NULL, IDC_ARROW);
    wcApp.hIcon        =LoadIcon(hInst, szIconName);
    wcApp.lpszMenuName =0;
    wcApp.hbrBackground=(HBRUSH) GetStockObject(WHITE_BRUSH);
    wcApp.style        =CS_HREDRAW|CS_VREDRAW;
    wcApp.cbClsExtra   =0;
    wcApp.cbWndExtra   =0;
    if (!RegisterClass (&wcApp))
        return 0;

    hWnd=CreateWindow(szProgName,"Flying Saucer Program",
                     WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,
                     CW_USEDEFAULT,CW_USEDEFAULT,
                     CW_USEDEFAULT, (HWND) NULL, (HMENU) NULL,
                     hInst, (LPSTR) NULL);
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
```



```
// load saucer bitmap
hBitmap=LoadBitmap(hInst,szBMName);
while (GetMessage(&lpMsg,0,0,0)) {
    TranslateMessage(&lpMsg);
    DispatchMessage(&lpMsg);
}
return(lpMsg.wParam);
}
LRESULT CALLBACK WndProc(HWND hWnd,UINT messg,
                        WPARAM wParam,LPARAM lParam)
{
    HDC hdc;
    HDC hmdc;
    RECT rcWnd;
    static BITMAP bm;
    switch (messg)
    {
        case WM_CREATE:
            // initial values
            xPosInit=200;
            yPosInit=200;
            xStep=1;
            yStep=1;
            xPos=xPosInit;
            yPos=yPosInit;
            iTimer=SetTimer(hWnd,1,10,NULL);
            break;
        case WM_TIMER:
            // with each timer tick, draw image
            hdc=GetDC(hWnd);
            GetClientRect(hWnd,&rcWnd);
            hmdc=CreateCompatibleDC(hdc);
            xPos+=xStep;
            yPos+=yStep;
            // draw image
            SelectObject(hmdc,hBitmap);
            GetObject(hBitmap,sizeof(bm),(LPSTR) &bm);
            BitBlt(hdc,xPos,yPos,bm.bmWidth,bm.bmHeight,
                hmdc,0,0,SRCCOPY);
            // check left and right window edges
            if((xPos+bm.bmWidth > rcWnd.right) ||
                (xPos < rcWnd.left)) {
```




```

        xStep=-xStep;
    }
    // check top and bottom window edges
    if((yPos+bm.bmHeight > rcWnd.bottom) ||
        (yPos < rcWnd.top)) {
        yStep=-yStep;
    }
    ReleaseDC(hWnd,hdc);
    DeleteDC(hmdc);
    break;
case WM_DESTROY:
    if (hBitmap) DeleteObject(hBitmap);
    if(iTimer) KillTimer(hWnd,1);
    PostQuitMessage(0);
    break;
default:
    return(DefWindowProc(hWnd,messg,wParam,lParam));
    break;
}
return(0);
}

```

图 8-14 给出了调试后飞碟在窗口中已经移动了若干次的结果。

在 Windows 应用程序中与边界有关的问题相当普遍，这就要求清楚 Windows 是如何使用窗口范围、视口以及客户区域。在跟踪这种类型的错误时，Debugger 可以是一个很重要的信息源。

8.3.2 使用鼠标绘画

下面的工程名称为 sketch，该工程包括源代码 sketch.cpp，与本章前面所讨论的程序非常相似。但是 sketch 工程还包含几种 Windows 资源，包括一个菜单和对话框。另外，初始提供的程序代码可以正确运行。在这一例子中，我们将对原始的工程做一些改动，以增强工程的一些特性。

资源为 Windows 应用程序增添了又一层的复杂性。同样，如果读者有一段时间没有编写过 Windows 代码，那么最好选一本处理面向过程的 Windows 编码方面的书看看。

在 Visual C++ 环境下创建一个名为 sketch 的 Win32 工程，在该空工程中添加如下的源代码文件，并命名为 sketch.cpp：

```

//
// (working - but not quite complete) -sketch.cpp
// Draws in the client area with the mouse.

```

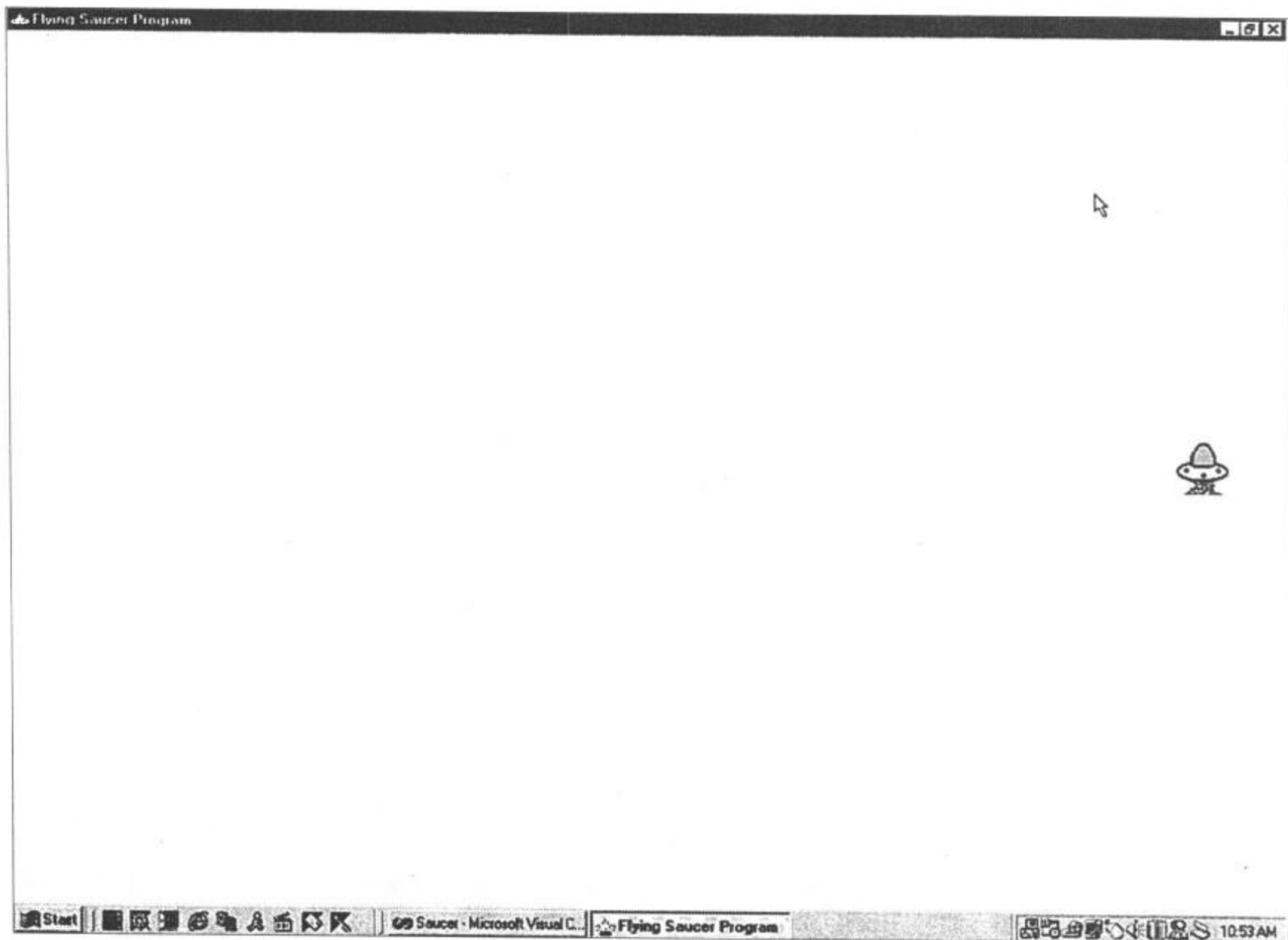


图 8-14 飞碟成功地穿越在窗口内

```
// Draws with color selected from list of predefined colors.
// Pen widths also selected from a list of predefined pens.
// Copyright (c) William H. Murray and Chris H. Pappas, 2000
//
#include <windows.h>
#include "resource.h"
HINSTANCE hInst;
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK AboutDlgProc(HWND, UINT, WPARAM, LPARAM);
char szProgName[]="ProgName";           // app name
char szAppName[]="SketchMenu";          // menu name
char szCursorName[]="SketchCursor";     // cursor name
char szIconName[]="SketchIcon";         // icon name
static WORD wColor;                      // color from menu
BOOL bDrawtrail;                         // (t/f) draw?
```



```

POINT omouselocat,nouselocat;    // position
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE hPreInst,
                  LPSTR lpszCmdLine,int nCmdShow)
{
    HWND hWnd;
    MSG lpMsg;
    WNDCLASS wcApp;

    wcApp.lpszClassName=szProgName;
    wcApp.hInstance    =hInst;
    wcApp.lpfnWndProc  =WndProc;
    wcApp.hCursor      =LoadCursor(hInst,szCursorName);
    wcApp.hIcon        =LoadIcon(hInst,szIconName);
    wcApp.lpszMenuName =szApplName;
    wcApp.hbrBackground=(HBRUSH) GetStockObject(WHITE_BRUSH);
    wcApp.style        =CS_HREDRAW|CS_VREDRAW;
    wcApp.cbClsExtra   =0;
    wcApp.cbWndExtra   =0;
    if (!RegisterClass (&wcApp))
        return 0;

    hWnd=CreateWindow(szProgName,"Drawing with the Mouse",
                     WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,
                     CW_USEDEFAULT,CW_USEDEFAULT,
                     CW_USEDEFAULT,(HWND) NULL,(HMENU) NULL,
                     hInst,(LPSTR) NULL);
    ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);
    while (GetMessage(&lpMsg,0,0,0)) {
        TranslateMessage(&lpMsg);
        DispatchMessage(&lpMsg);
    }
    return(lpMsg.wParam);
}
// About dialog box control selection
BOOL CALLBACK AboutDlgProc(HWND hDlg,UINT messg,
                          WPARAM wParam,LPARAM lParam)
{
    switch (messg) {
        case WM_INITDIALOG:
            break;
        case WM_COMMAND:
            switch (LOWORD(wParam)) {

```



```
        case IDOK:
            EndDialog(hDlg, 0);
            break;
        default:
            return FALSE;
    }
    break;
default:
    return FALSE;
}
return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT messg,
                        WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    HMENU hmenu;
    static HPEN hOpen, hNPen;
    static COLORREF tempcolor=RGB(0,0,0);
    static COLORREF wColorValue[5]={RGB(0,0,0),          //BLACK
                                     RGB(255,255,255),    //WHITE
                                     RGB(255,0,0),         //RED
                                     RGB(0,255,0),         //GREEN
                                     RGB(0,0,255),         //BLUE

    static int penwidth=2;
    static POINT pt;
    switch (messg)
    {
        case WM_COMMAND:
            // menu item selections
            switch (LOWORD(wParam))
            case ID_OPTIONS_CLEAR:
                tempcolor=wColorValue[1];
                InvalidateRect(hWnd, NULL, TRUE);
                break;
            case ID_OPTIONS_EXIT:
                SendMessage(hWnd, WM_CLOSE, 0, 0L);
                break;
            case ID_PEN_TWO:
                penwidth=2;
                break;
            case ID_PEN_FIVE:
```



```

        penwidth=5;
        break;
case ID_PEN_TEN:
    penwidth=10;
    break;
case ID_PEN_THIRTY:
    penwidth=30;
    break;
case ID_PEN_SIXTY:
    penwidth=60;
    break;
case ID_HELP_ABOUT:
    DialogBox((HINSTANCE) GetModuleHandle(NULL),
        "AboutDlgBox",hWnd,
        AboutDlgProc);
    break;
case IDM_BLACK:
case IDM_WHITE:
case IDM_RED:
case IDM_GREEN:
case IDM_BLUE:
    hmenu=GetMenu(hWnd);
    CheckMenuItem(hmenu,wColor,MF_UNCHECKED);
    wColor=LOWORD(wParam);
    CheckMenuItem(hmenu,wColor,MF_CHECKED);
    tempcolor=wColorValue[wColor-IDM_BLACK];
    break;
    default:
    break;
}
break;
case WM_LBUTTONDOWN:
    // draw when mouse button is down
    nmouselocat.x=LOWORD(lParam);
    nmouselocat.y=HIWORD(lParam);
    omouselocat=nmouselocat;
    SetCapture(hWnd);
    bDrawtrail=TRUE;
    break;
case WM_MOUSEMOVE:
    // follow the mouse around
    if (bDrawtrail) {
        omouselocat=nmouselocat;
    }

```



```
        nmouselocat.x=LOWORD(lParam);
        nmouselocat.y=HIWORD(lParam);
        InvalidateRect(hWnd, NULL, FALSE);
        UpdateWindow(hWnd);
    }
    break;
case WM_LBUTTONDOWN:
    // do not draw when mouse button is up
    ReleaseCapture();
    bDrawtrail=FALSE;
    break;
case WM_PAINT:
    hdc=BeginPaint(hWnd, &ps);
    hNPen=CreatePen(PS_SOLID, penwidth, tempcolor);
    hOPen=(HPEN) SelectObject(hdc, hNPen);
    MoveToEx(hdc, omouselocat.x, omouselocat.y, NULL);
    LineTo(hdc, nmouselocat.x, nmouselocat.y);
    SelectObject(hdc, hOPen);
    DeleteObject(hNPen);

    ValidateRect(hWnd, NULL);
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return(DefWindowProc(hWnd, messg, wParam, lParam));
    break;
}
return(0);
}
```

该工程也将使用到一个名为 Sketch.rc 的资源脚本文件，其中包含了有关资源如光标、对话框、图标及菜单的详细描述。在本应用程序中，我们使用了一个外形类似于一个画笔刷的光标，如图 8-15 所示。

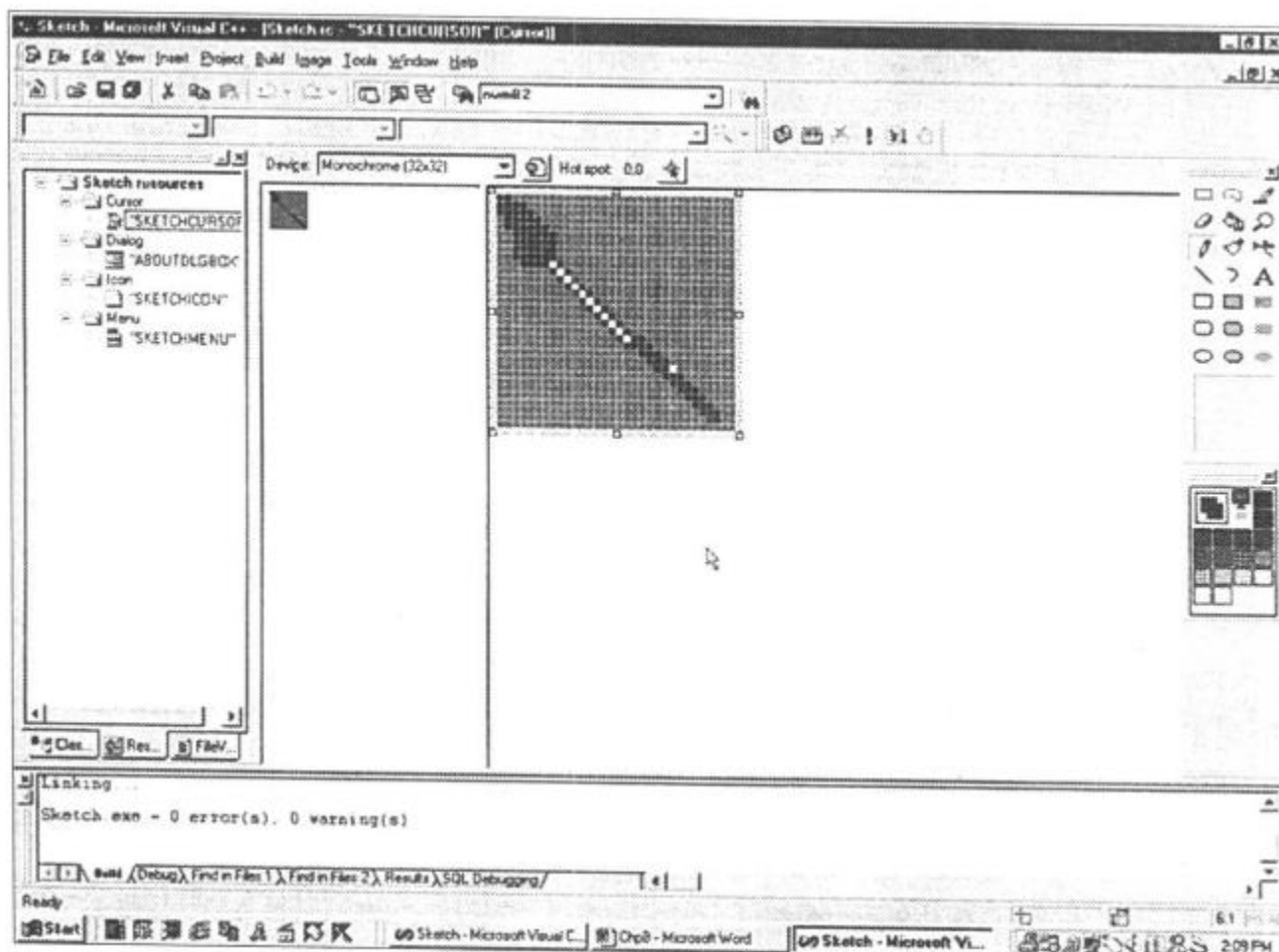


图 8-15 为 sketch 工程创建的一个独特光标

About 对话框用来增强该应用程序，其中给出了关于工程创建者的信息，图 8-16 所示为一个简单的 About 对话框。

在资源编辑器中设计了一个 32×32 的图标，如图 8-17 所示。

该应用程序运行时，可以在标题栏区域的左上角看到这一图标。

这一程序使用了四个菜单，Pen-Colors 菜单可在图 8-18 中看到，其中给出了五种颜色选择。

其他的菜单包含了关于清除窗口、设置画笔宽度或调用 About 对话框等菜单选项。

当在资源编辑器中将一个菜单项添加到一个菜单时，用户有一个设置唯一 ID 值的机会。图 8-19 给出了为 Green 菜单项输入的 ID 值。

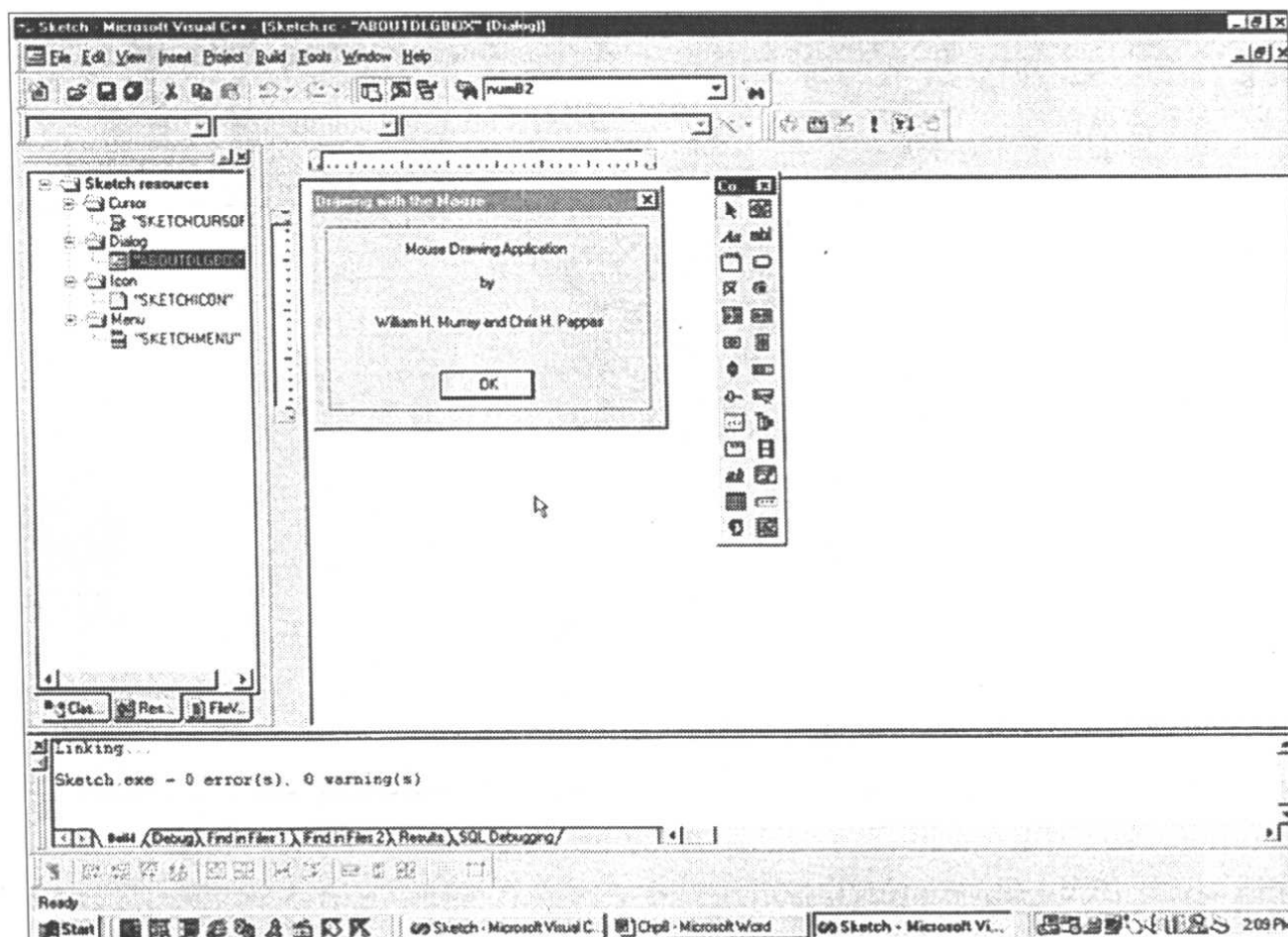


图 8-16 About 对话框为该工程增加了一些专业品味

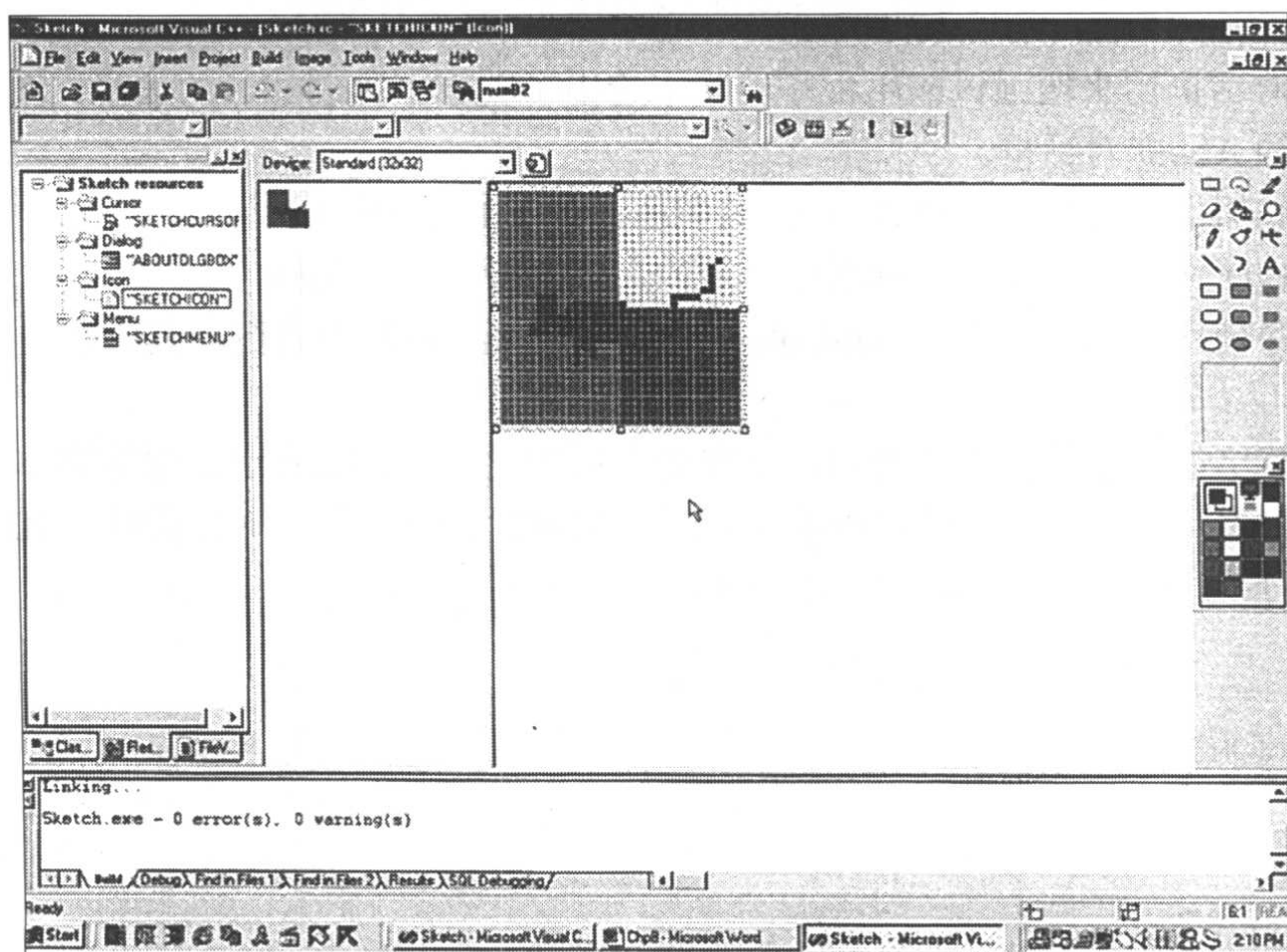


图 8-17 为该工程创建一个独特图标

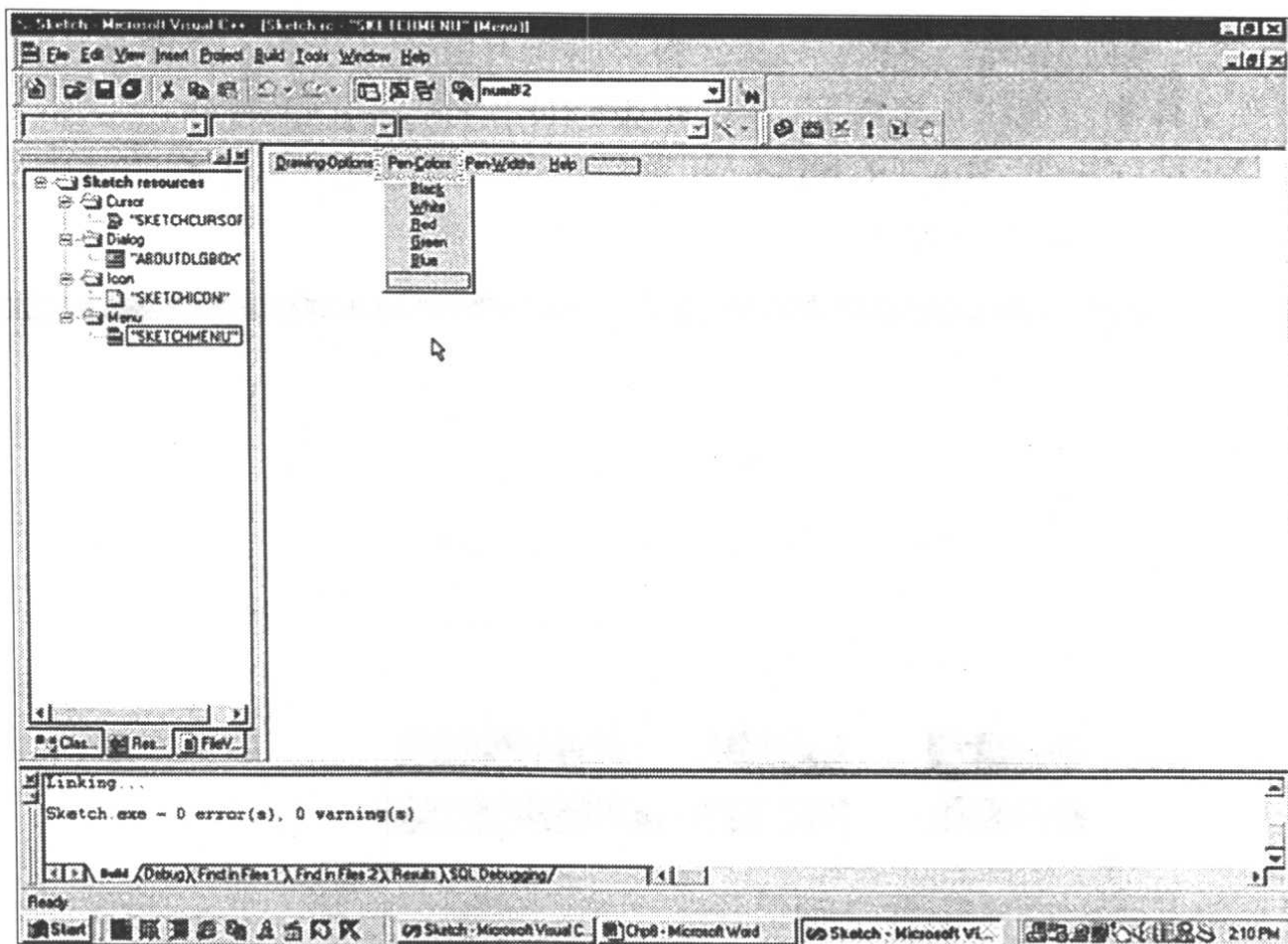


图 8-18 工程初始使用的由五种绘画颜色组成的调色板

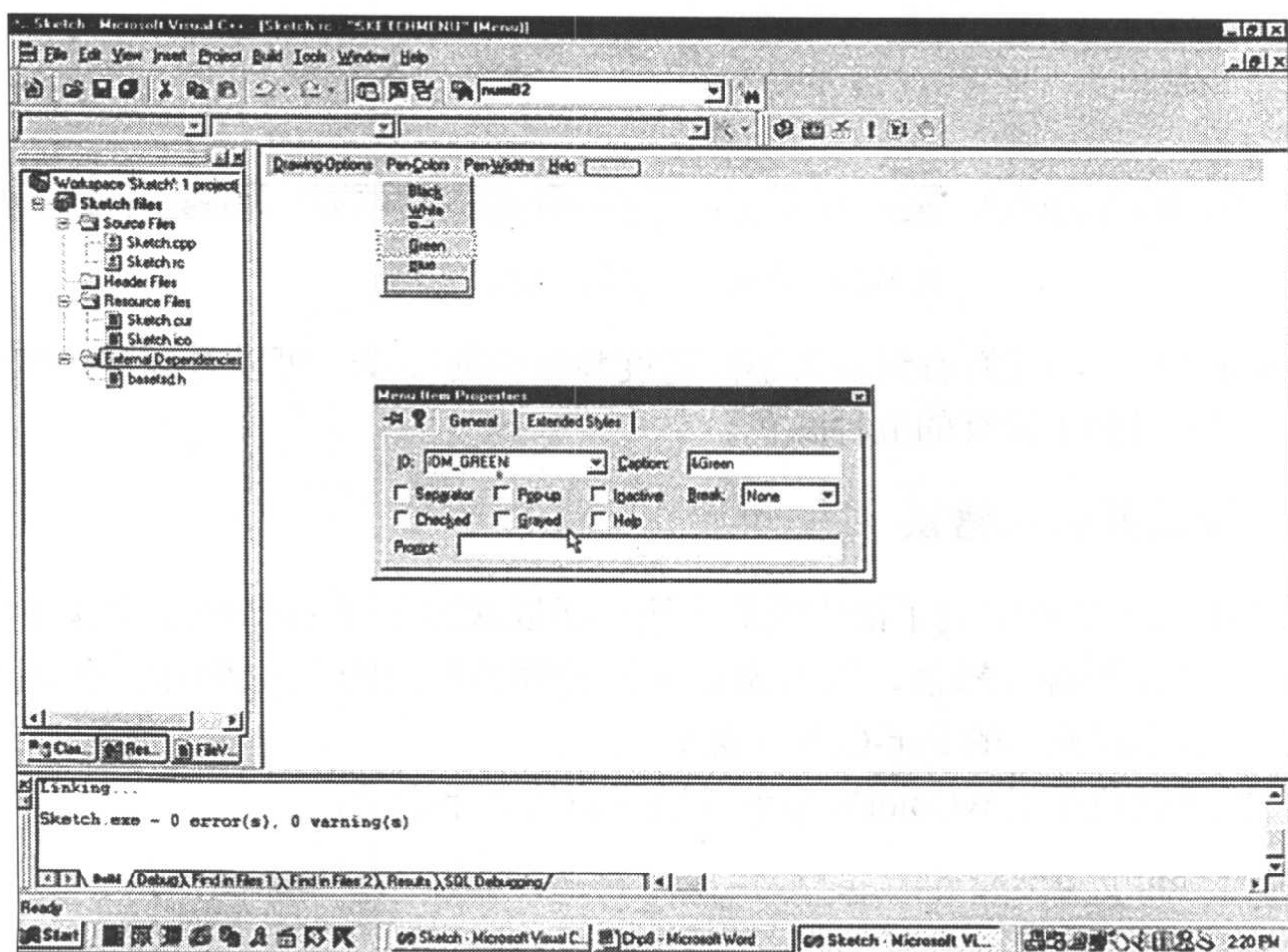


图 8-19 与 Green 菜单项关联的唯一 ID 值



所有的其他菜单项也都有一个应用于它的类似 ID 值, 这些 ID 值保存在该工程所包含的 resource.h 头文件中。在添加资源时, 将自动将 ID 值添加到该文件中。

通过选择 Visual C++ 的 Build|Rebuild All 菜单选项, 可以建立该工程的应用程序。该工程的执行情况如图 8-20 所示。

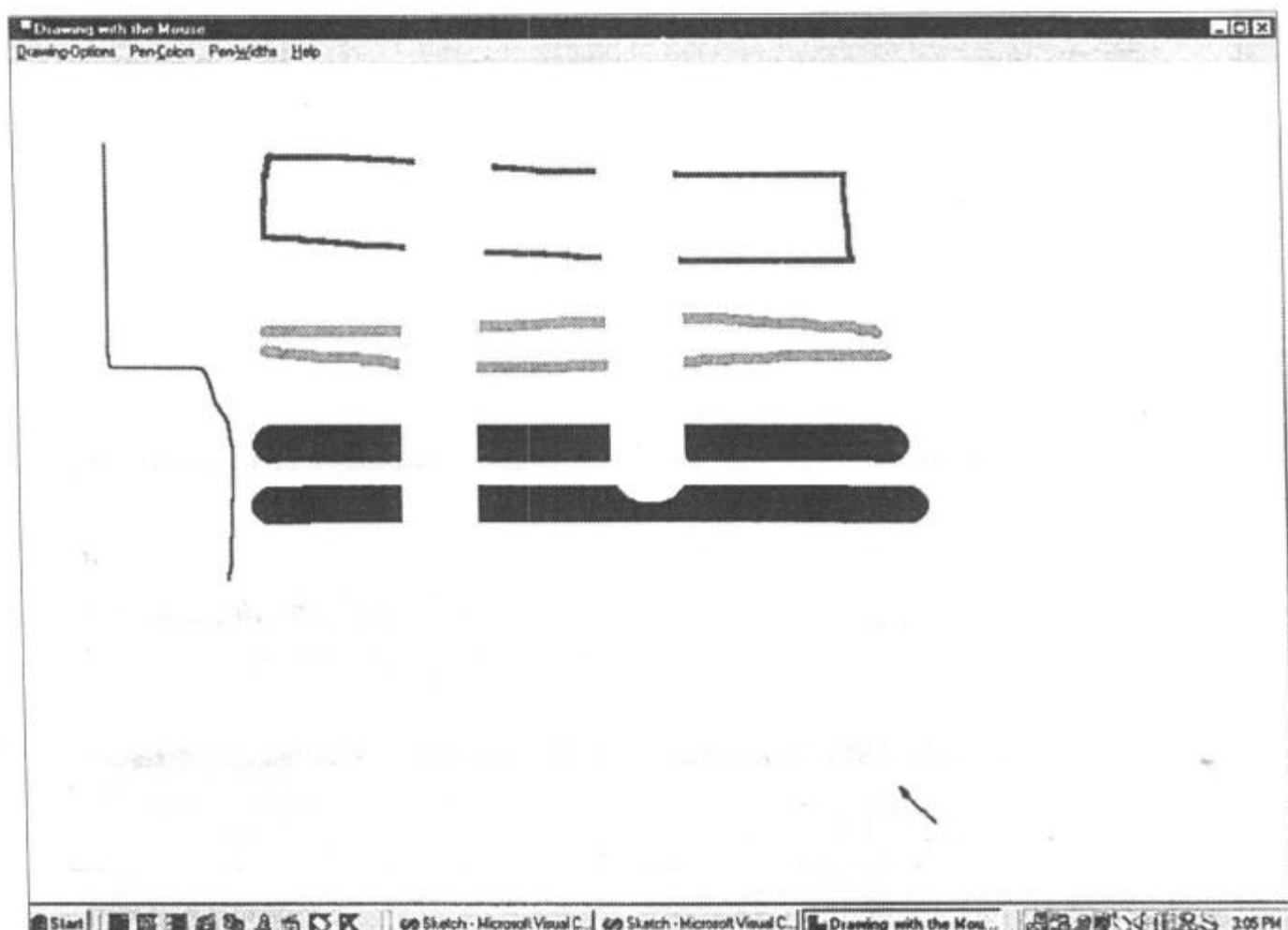


图 8-20 sketch 工程第一版的试运行

在检查图 8-20 时, 我们看到多条不同宽度和颜色的线段。可以看到这些宽度的变化, 但也必须知道它使用到了所有的五种颜色。

8.3.2.1 修改并引入错误

我们假设 sketch 工程的到了很广泛的应用, 所以决定在 Pen-Colors 菜单中添加额外的七种绘画颜色。使用资源编辑器, 以与添加前五种颜色相同的方式添加新的颜色。图 8-21 显示了资源编辑器中扩充后的 Pen-Colors 菜单。

下一步, 扩展颜色数组 wColorValue[] 以适应新的菜单选项:

```
static COLORREF wColorValue[12]={RGB(0,0,0),           //BLACK
                                   RGB(255,255,255),      //WHITE
                                   RGB(255,0,0),           //RED
                                   RGB(255,96,0),          //ORANGE
```

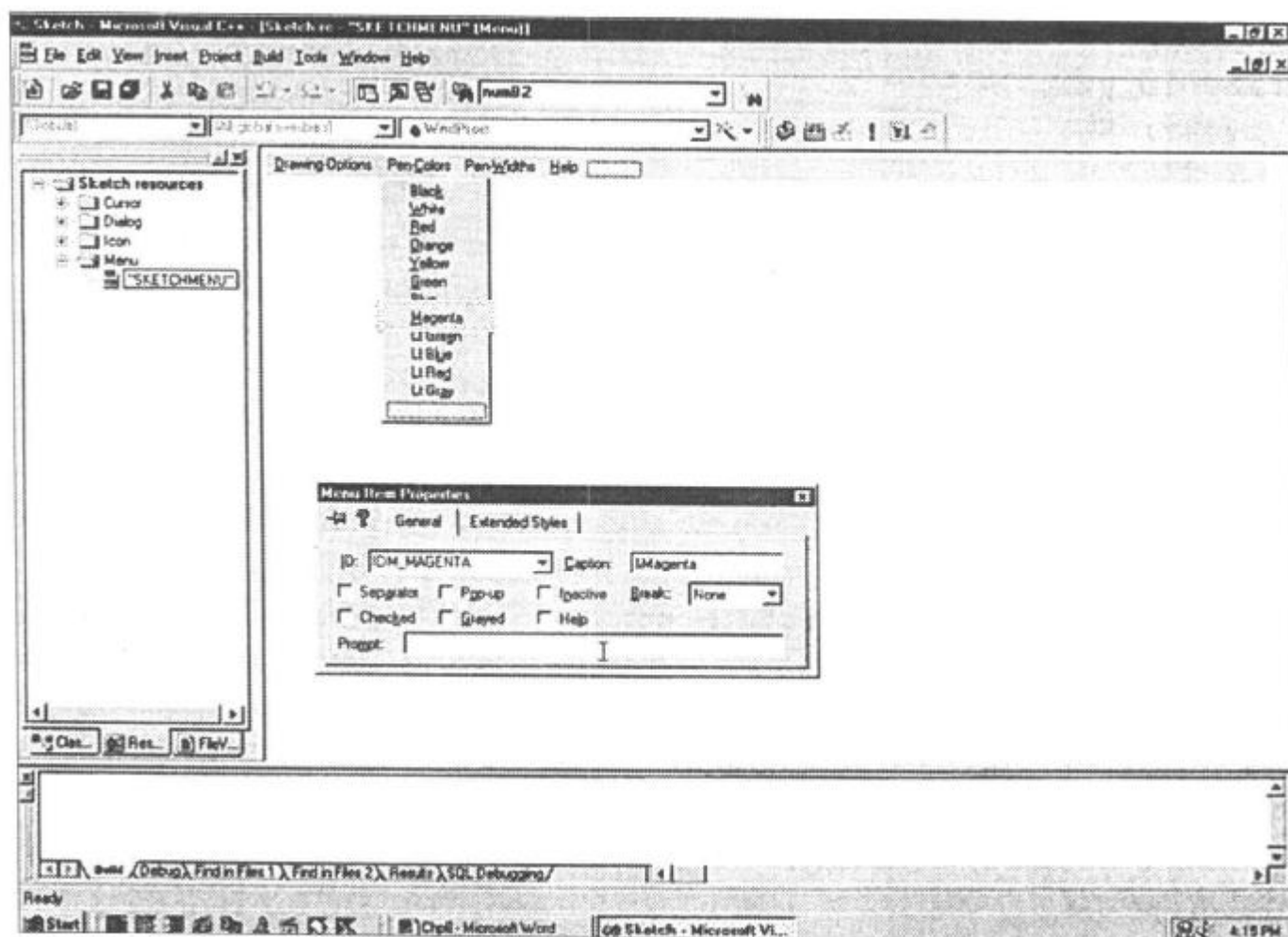


图 8-21 七种新的颜色选项添加到了 Pen-Colors 菜单中

```

RGB(255,255,0),           //YELLOW
RGB(0,255,0),             //GREEN
RGB(0,0,255),             //BLUE
RGB(255,0,255),           //MAGENTA
RGB(128,255,0),           //LT GREEN
RGB(0,255,255),           //LT BLUE
RGB(255,0,159),           //LT RED
RGB(180,180,180));       //LT GRAY
    
```

最后，扩展处理 WM_COMMAND 消息的 case 语句，所做的修改以黑体表示：

```

case WM_COMMAND:
    // menu item selections
    switch (LOWORD(wParam)) {
        case ID_OPTIONS_CLEAR:
            tempcolor=wColorValue[1];
            InvalidateRect(hWnd,NULL,TRUE);
            break;
        case ID_OPTIONS_EXIT:
            SendMessage(hWnd,WM_CLOSE,0,0L);
            break;
    }
    
```



```
case ID_PEN_TWO:
    Penwidth=2;
    break;
case ID_PEN_FIVE:
    Penwidth=5;
    Break;
case ID_PEN_TEN:
    Penwidth=10;
    Break;
case ID_PEN_THIRTY:
    Penwidth=30;
    Break;
case ID_PEN_SIXTY:
    Penwidth=60;
    Break;
case ID_HELP_ABOUT:
    DialogBox((HINSTANCE) GetModuleHandle(NULL),
        "AboutDlgBox",hWnd,
        AboutDlgProc);

    break;
case IDM_BLACK:
case IDM_WHITE:
case IDM_RED:
case IDM_ORANGE:
case IDM_YELLOW:
case IDM_GREEN:
case IDM_BLUE:
case IDM_MAGENTA:
case IDM_LTGREEN:
case IDM_LTBLUE:
case IDM_LTRD:
case IDM_LTGRAY:
    hmenu=GetMenu(hWnd);
    CheckMenuItem(hmenu,wColor,MF_UNCHECKED);
    wColor=LOWORD(wParam);
    CheckMunuItem(hmenu,wColor,MF_CHECKED);
    tempcolor=wColorValue[wColor-IDM_BLACK];
    break;
default:
    break;
}
break;
```



现在，编译该工程并测试该应用程序。可以看到什么？虽然我们有 12 种可以选择的颜色，但是颜色并不正确。事实上，颜色被搞得一团糟。例如，橙黄色变成了绿色，黄色变成了兰色，等等。这些错误是如何进入应用程序中的？我们的第一反应是为某些新的颜色所指定的 RGB 值不正确。这也常常是这一类问题的根源。但在此工程中不是这样。除非读者以前已经遇到过这种问题，否则这是一个非常难以发现的问题。现在又是使用 Debugger 的时候了。

8.3.2.2 查找并修复错误

以一般方式启动一个远程调试器会话，在远程目标计算机(sony)上打开 Pen-Colors 菜单。然后选择 black(黑色)笔，如图 8-22 所示。

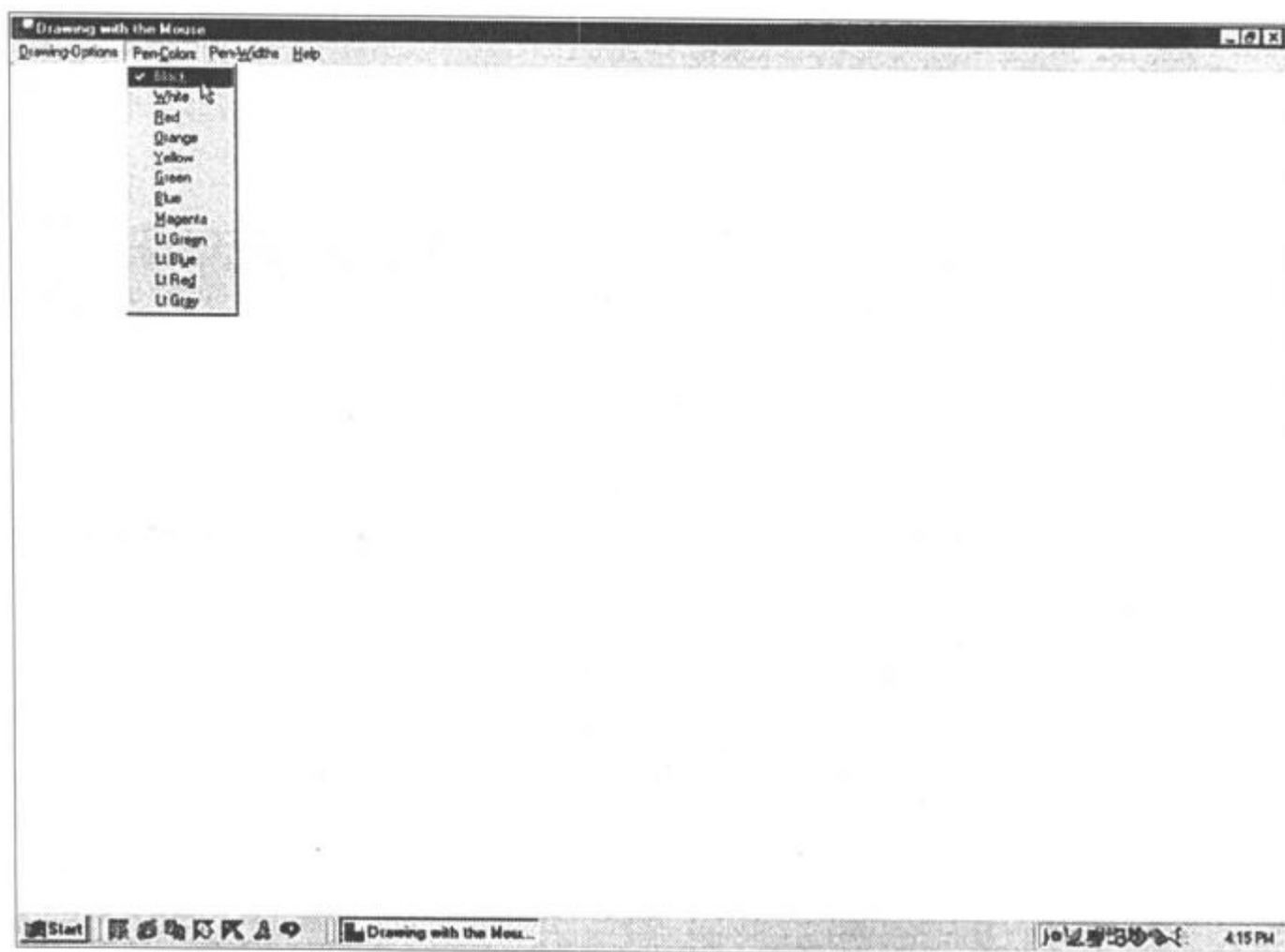


图 8-22 从 Pen-Colors 菜单中选择 black 颜色

在如图 8-23 所示的代码处设置一个断点，将 *wColor* 变量添加到 Watch 窗口中，然后执行代码到该断点。

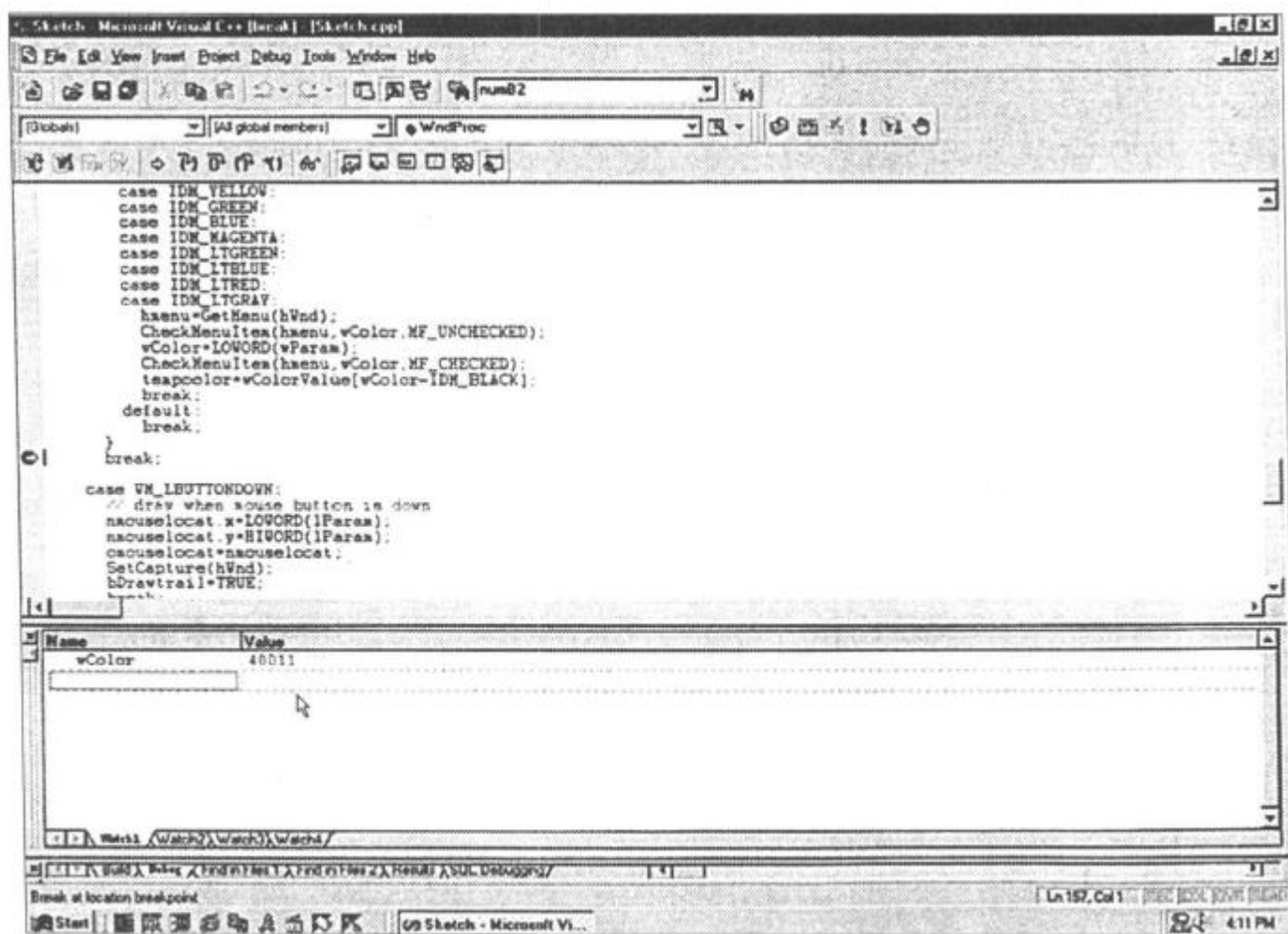


图 8-23 将 wColor 值放到 Watch 窗口中

wColor 变量的值将包含一个整型值，此处为 40011。

现在重复这一过程，选择每一个菜单项，运行到断点处，并记录 *wColor* 的值。表 8-1 总结了所获得的结果。

表 8-1 Debugger 返回的 wColor 值

返回的整数	所需的颜色
40011	Black(黑色)
40012	White(白色)
40013	Red(红色)
40014	Orange(橙黄色)
40015	Yellow(黄色)
40016	Green(绿色)
40017	Blue(蓝色)
40018	Magenta(洋红色)
40019	Light green(淡绿色)
40020	Light blue(淡蓝色)
40021	Light red(淡红色)
40022	Light gray(淡灰色)



仔细研究这一表，注意原始颜色的顺序：40011、40012、40013、40014 及 40015。而七种新颜色的返回值为：40016、40017、40018、40019、40020、40021 及 40022。

这一程序选择颜色时使用了一种有效的处理方法，但这依赖于菜单项的顺序。我们在稍后将解释这一点。当创建初始表后，Windows 为每一个菜单选项的 ID 值指定顺序标识常量，即为 `IDM_BLACK` 指定 40011，而为 `IDM_RED` 指定 40012，依此类推。然而，当我们回过头来添加新项目时，特别是添加橙黄色和绿色时，将从上次的菜单项号(40015)继续这一顺序。这样，为 `IDM_ORANGE` 指定的是 40016，依此类推。

然而，此处的有效代码片段希望针对于每一个菜单项的顺序号从顶部开始至底部结束：

```
tempcolor=wColorValue[wColor-IDM_BLACK];
```

数组 `wColorValue` 的索引是由 `wColor` 中返回的值减去与 `IDM_BLACK` 关联的常量值决定的。例如，如果选择 Red 菜单项，结果为 `40013-40011=2`。索引值 2 正确地指向了 `wColorValue[]` 中的红色值 `RGB(255, 0, 0)`。作为另一个例子，对 Orange 菜单选项执行该运算。可以看出，它并未正确指向数组值。快速修复这一问题的唯一方法是，进入 `resource.h` 头文件，调整这些值。

下面是 `resource.h` 头文件中修改了 ID 值的部分：

```
#define IDM_BLACK          40011
#define IDM_WHITE          40012
#define IDM_RED            40013
#define IDM_GREEN          40016
#define IDM_BLUE           40017
#define IDM_ORANGE         40014
#define IDM_YELLOW         40015
#define IDM_MAGENTA        40018
#define IDM_LTGREEN        40019
#define IDM_LTBLUE         40020
#define IDM_LTRED          40021
#define IDM_LTGRAY         40022
```

24×7

当在资源编辑器中添加菜单项时，顺序的整数值指定给与每一个菜单项对应的 ID 值。作为 Windows 程序员的一种传统，使用这些有序数字可以帮助减少程序代码中的多余的 `case` 语句，`case` 语句本身就是一个经常出现程序错误的源。

对按照顺序指定的整数值的的要求非常简单：

- 首先，菜单项的序列应该是相关的——例如，刷子颜色、画笔长度、背景颜色、线段长度等等。
- 其次，顺序值通常用于计算一个实际数据值数组的索引。这样，索引值即为通过计算从菜单中选择时所返回的项，减去指定给列表中第一个项的整数值。



- 最后，菜单项序列必须与 `resource.h` 头文件中的 ID 值序列及带有索引的数组值的顺序相匹配。

除了使用 Debugger 之外，还有一条好的建议，即设计菜单时应该事先仔细设计。然后当输入菜单项时，总是由 Windows 指定顺序值。

现在，表中的整数按顺序与颜色的位置相匹配。再次编译并测试该应用程序，现在，该应用程序运行正常。

对于菜单列表使用顺序整数值是一种流行的技术，这种方法对于处理相关项目的长列表很有效。但是，如果以前从未使用过的话，将带来一些必须解决的问题。在这一例子中 Debugger 证明了其价值。

8.4 小结

本章引入了一些强有力的调试技术，这些技术非常适用于 Windows 应用程序。首先，我们学会了如何利用一台远程目标计算机完成调试的功能。然后在使用面向过程的编程环境的同时，我们学习了如何检测、诊断及修复一些有趣的边界问题。我们学习这些边界问题，主要是针对那些不能小心区分窗口长度、客户区域长度等之间的区别的程序员而言的。在第二个例子中，我们看到了由于不理解创建菜单时 Windows 如何为 ID 值，如 `IDM_BLACK` 和 `IDM_ORANGE` 赋值，以及为工作代码引入错误是何等地容易。 ■

第三部分

DEBUGGING

DEBUGGING

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

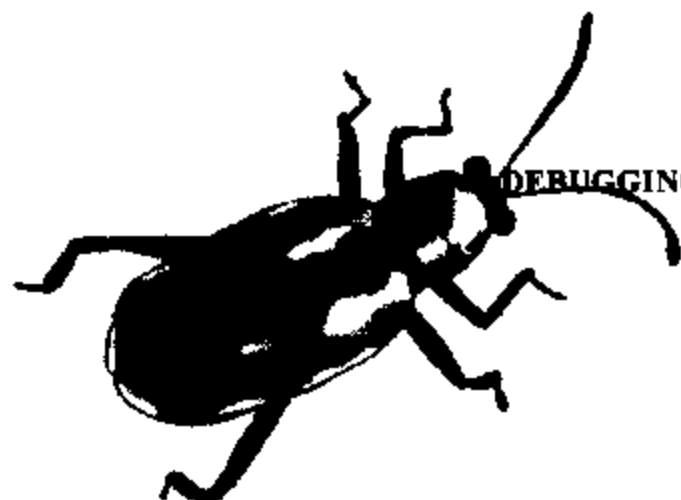
DEBUGGING

面向对象过程的环境

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

- 第 9 章 定位、分析和修复命令行代码错误
- 第 10 章 调试内联汇编语言代码
- 第 11 章 在 Windows 代码中定位、分析和修复错误

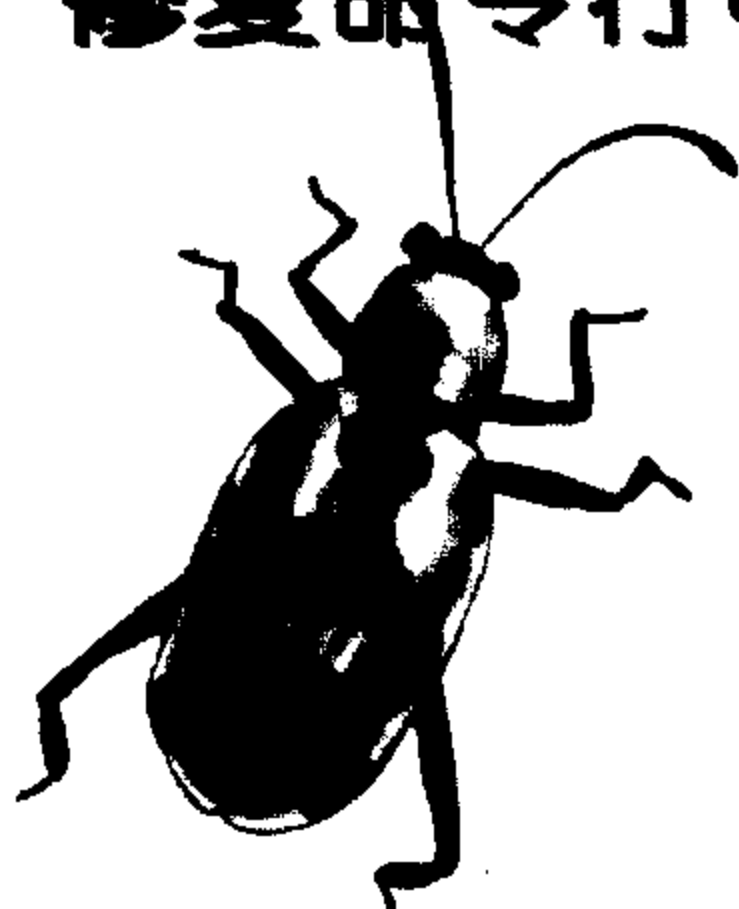
DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING



DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

第 9 章

定位、分析和
修复命令行中的错误





当从 C/C++ 过程程序设计转向面向对象设计时，必然增加了调试阶段的复杂性。Visual C++ 为此做好了准备，利用附加的调试命令、功能和前面讨论过的 Debugger 功能的高级用法。本章将探索这些新功能和学习如何管理其报告的信息。

9.1 高级调试工具

因为 Microsoft 的 Visual C++ Debugger 是一个非常全面的工具，因此掌握它要花些时间。本节将通过学习如何明确地指定 Debugger 完成具体的任务，继续磨练调试技巧。

9.1.1 内存卸出

在前面的章节中，已经看到 Variable 窗口和 Watch 窗口是如何出色地报告变量当前内容的。然而，当数据结构的复杂性增加时，内存卸出提供了更有意义的变量内容显示方式。

图 9-1 显示一个程序声明了一个有 1000 个元素的整型数组 *iArray*，初始化每个元素为其相应的下标。从图 9-1 可以看出，在 `main()` 程序的结尾括号 “}” 处设置了一个断点。只要按 F5 键就启动了 Debugger，执行 *iArray* 数组声明，完成 1000 次 for 循环。注意在图 9-1 中，光标停留在 for 循环体内使用的 *iArray* 数组处，显示出数组的开始地址为 0x0065f628。这一数组过大，在 Variables 窗口和 Watch 窗口中不易查看。但是，在图 9-2 中显示的 Memory 窗口中，可以很容易地处理这样大的对象。

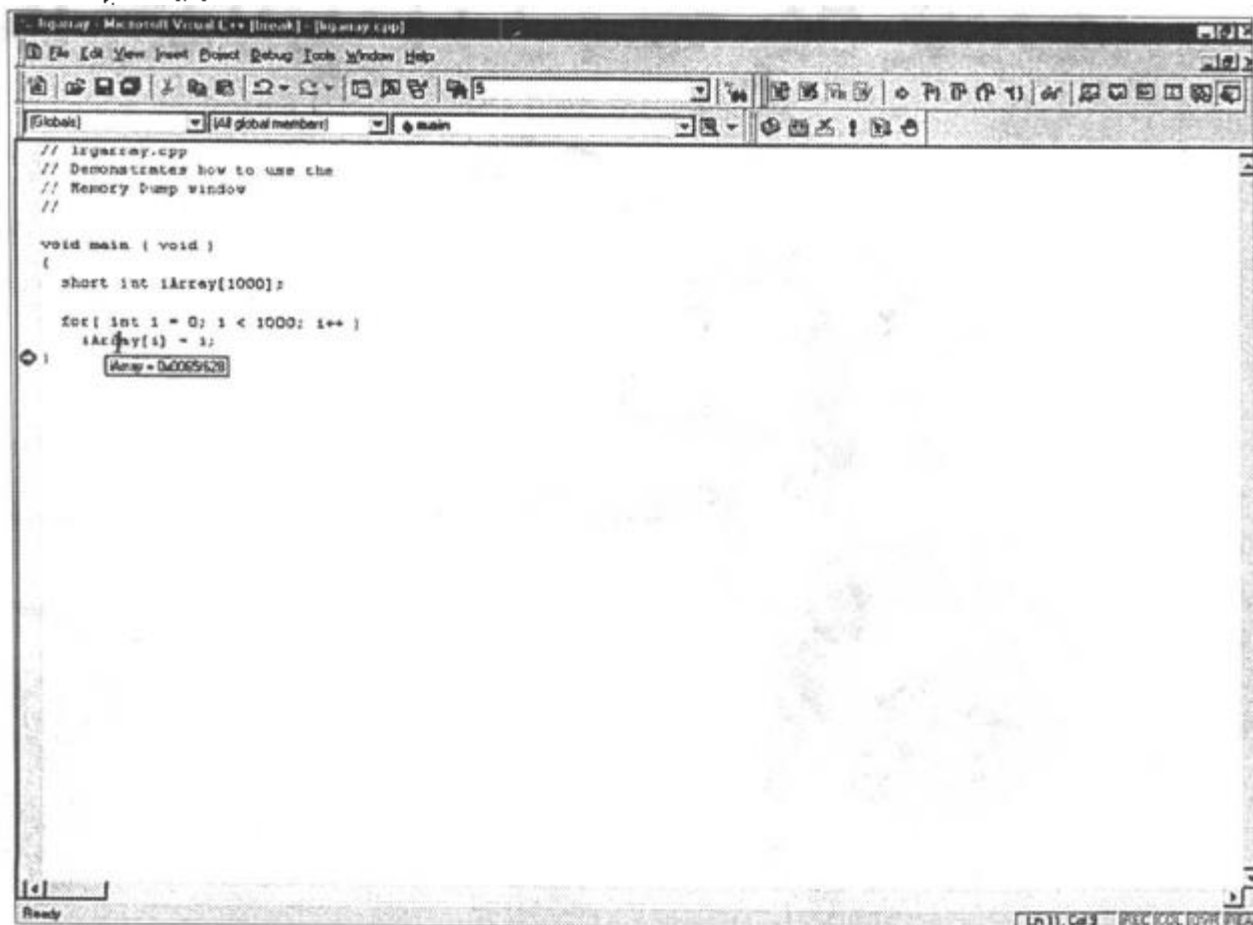


图 9-1 Debugger 显示 *iArray* 的物理地址

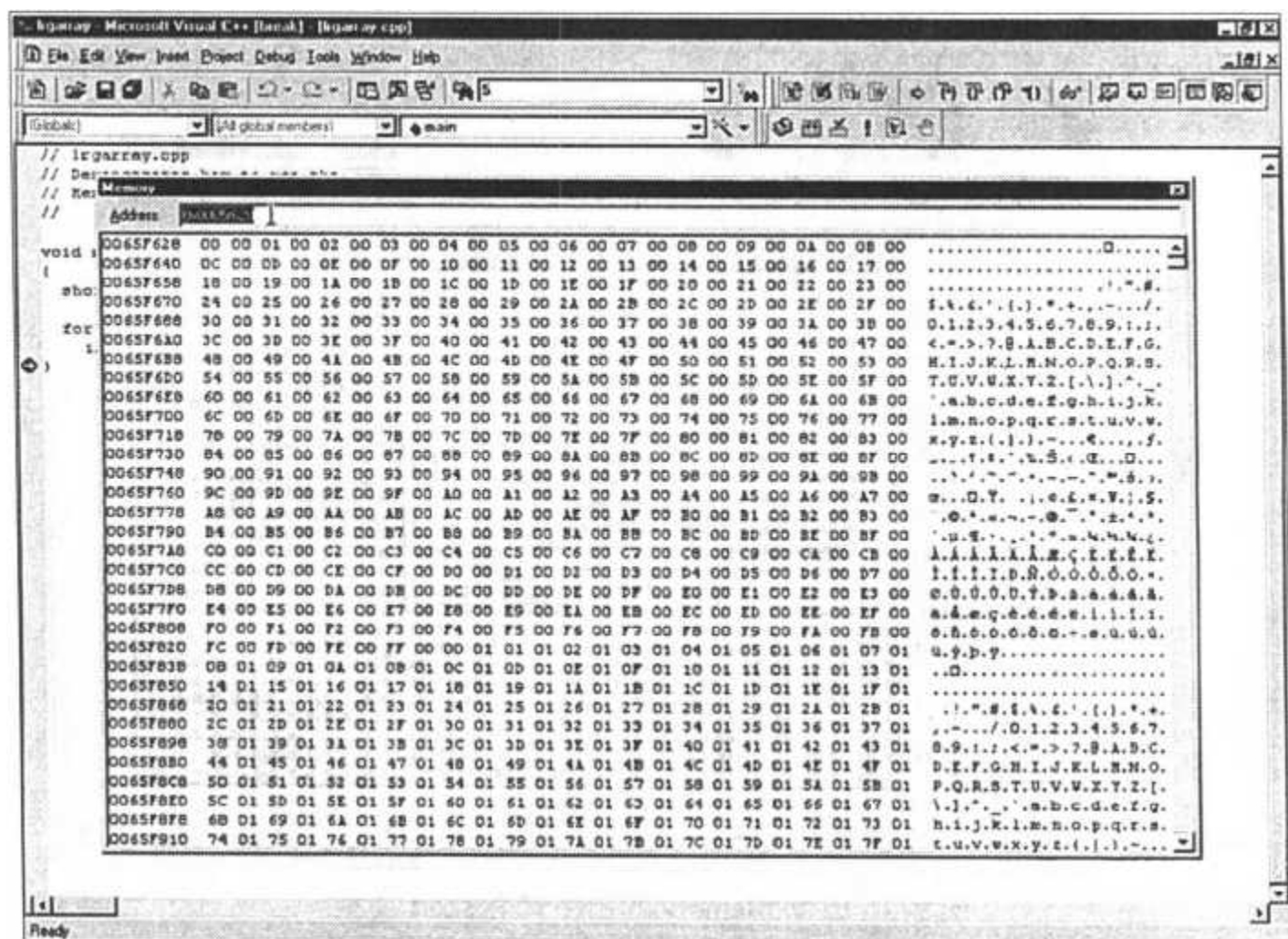


图 9-2 缺省的 Memory 窗口格式

为打开 Memory 窗口，可以选择 View | Debug Windows | Memory 选项，或按 ALT+6。图 9-2 显示了一个展开的 Memory 窗口，在其 Address 编辑框中输入了正确的地址。唯一的问题是在缺省情况下 Debugger 使用字节格式卸出。为改变这种显示格式，必须使用 Tools | Options 菜单，单击 Options 窗口的 Debug 标签，改变 Memory 窗口的输出格式(参见图 9-3)。

图 9-4 显示了调整后更有意义的显示格式。

从打开的 Memory 窗口，可以容易地操作滚动条查看整个对象的全部内容。尽管这是一个简单的例子，但其包含的思想对于调试更复杂的数据结构是非常有价值的。

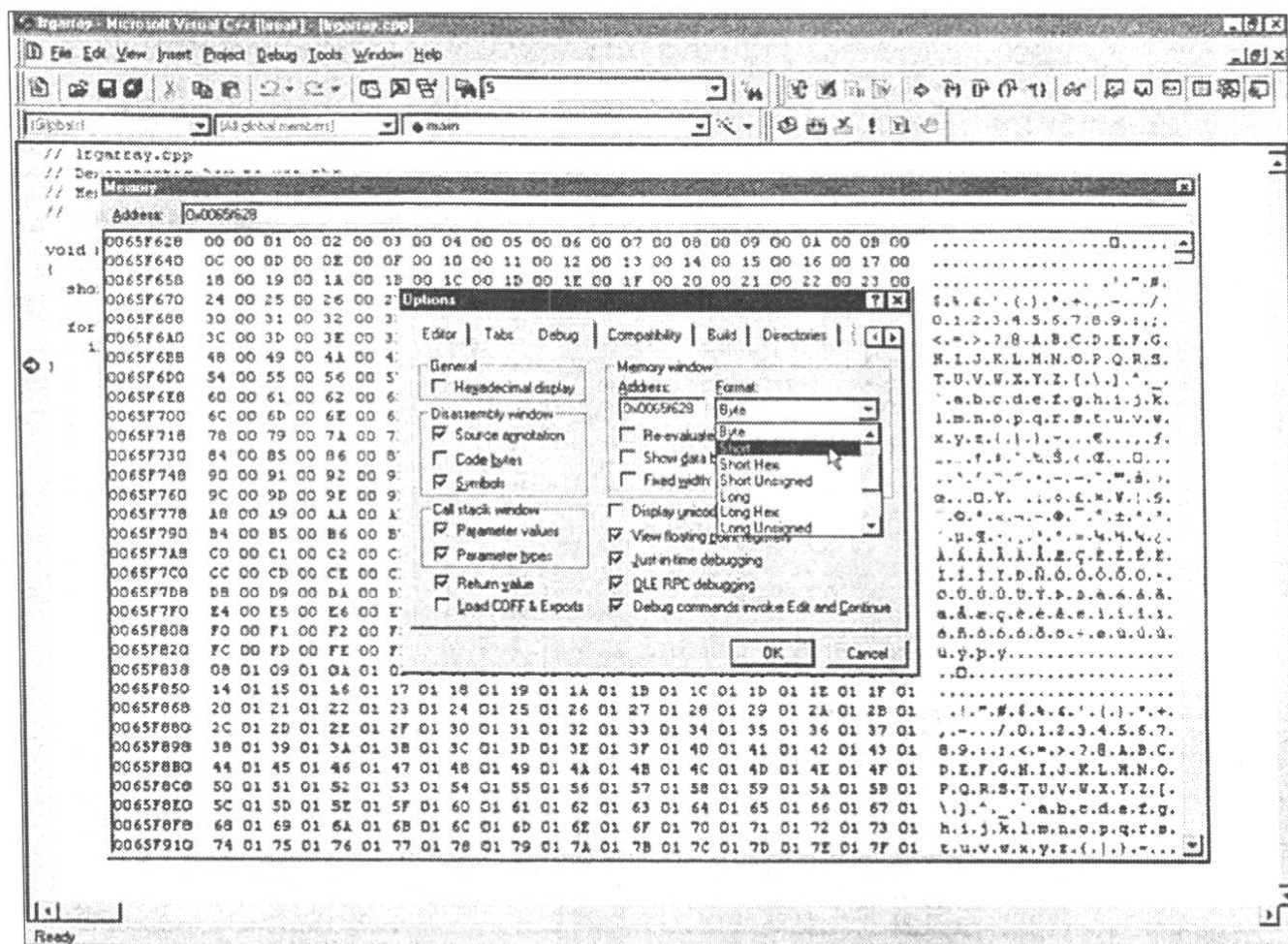


图 9-3 改变 Memory 卸出格式为 short 类型

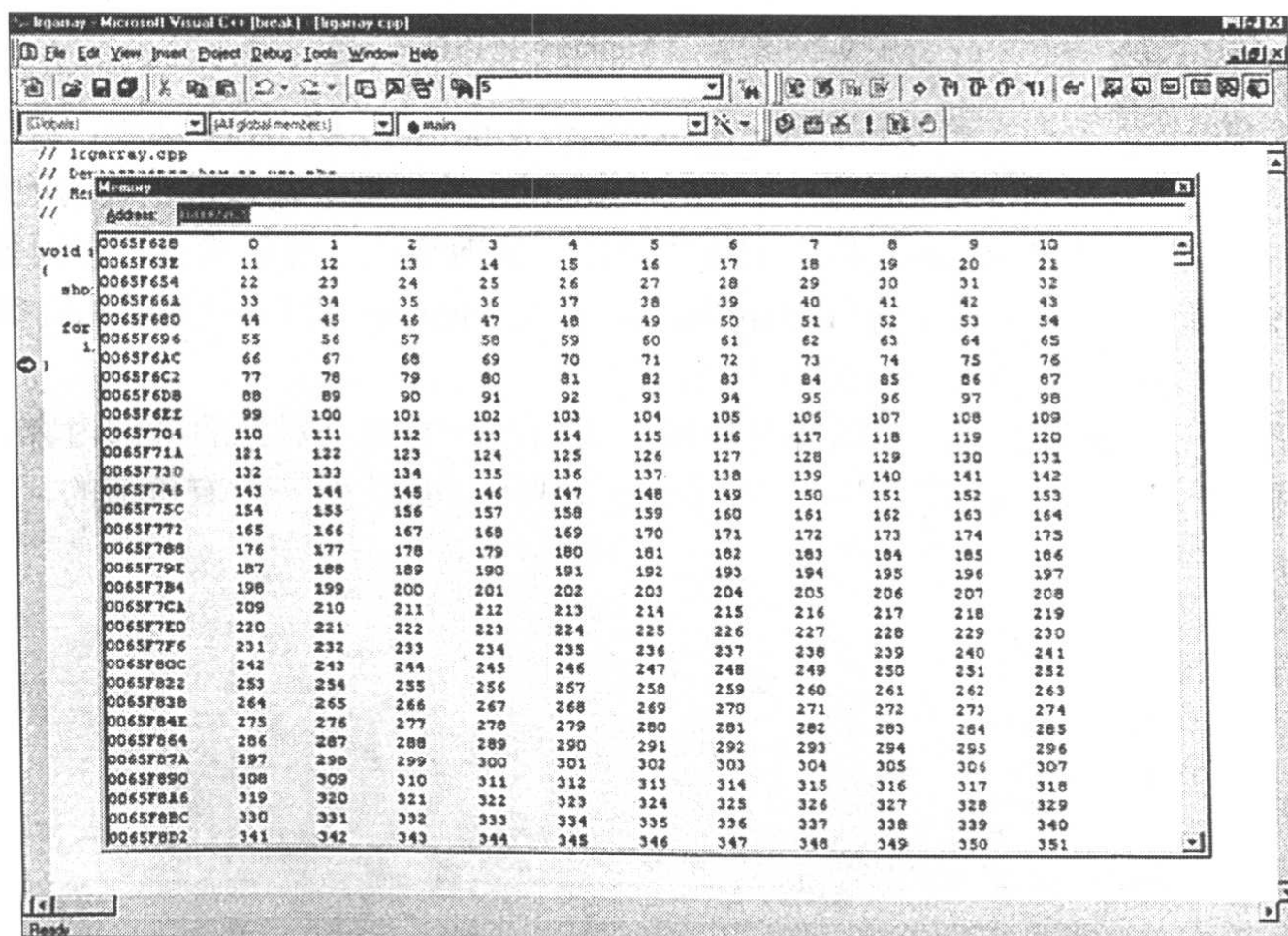


图 9-4 修改后的 Memory 窗口格式

9.1.2 定位错误参数从何处传递而来

设想编写了一个子程序(函数或方法),我们要成百上千次调用该子程序。通过对前面讨论过的调试命令的理解和使用,可以知道传递过来的哪个值是不正确的。但是我们没有线索,不知道这一错误调用从何处传递而来。在这种情况下,我们需要使用 Breakpoints Condition 选项。

在源程序代码中设置条件断点比设置无条件断点要稍微复杂些。要设置条件断点,首先单击 Visual C++ Edit 菜单,接着选择 Breakpoints(ALT+F9)选项,如图 9-5 所示。

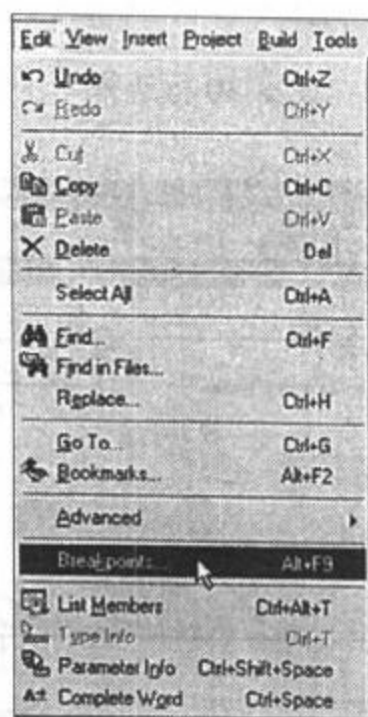


图 9-5 选择 Edit | Breakpoints 选项

条件 Breakpoints 窗口如图 9-6 所示,缺省时选中的是窗口中的 Location 标签。单击 Break at 编辑列表右边的向右箭头,显示出两个附加的定位选项,Line 和 Advanced。在图 9-6 中看到的 Line 30 是由 Visual C++ 自动输入的,因为在打开条件断点窗口前已经将“I”型光标设置在源程序的第 30 行(见本章后面的图 9-8),这是一个很好的自动输入捷径。

设计提示

当在特定的程序语句上设置条件断点时,首先将“I”型光标放在要设置条件断点的源代码语句上。接下来,使用 Edit | Breakpoints 选项或 ALT+F9,激活条件 Breakpoints 窗口,单击 Break at 编辑列表的向右箭头,再选定缺省行号即可。

对于本例来说 Line 30 是个理想的选择,因为它位于接收错误参数值的子程序内。一旦选择了 Line 30,暗淡显示的 Condition 按钮即被激活。单击 Condition 按钮显示 Breakpoints Condition 窗口,如图 9-7 所示。

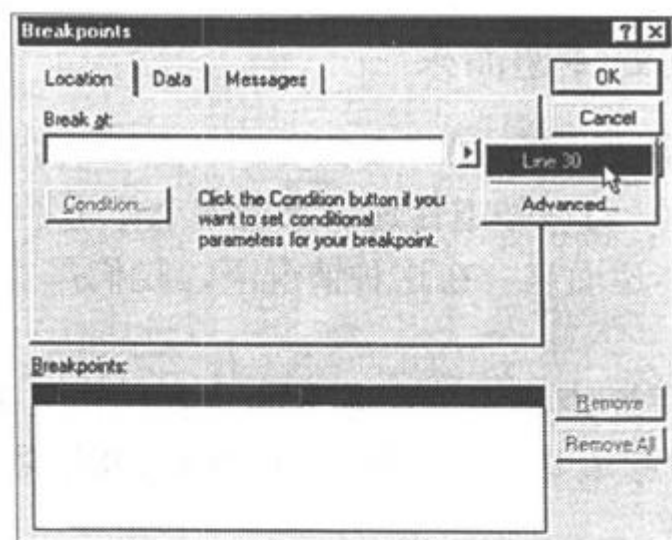
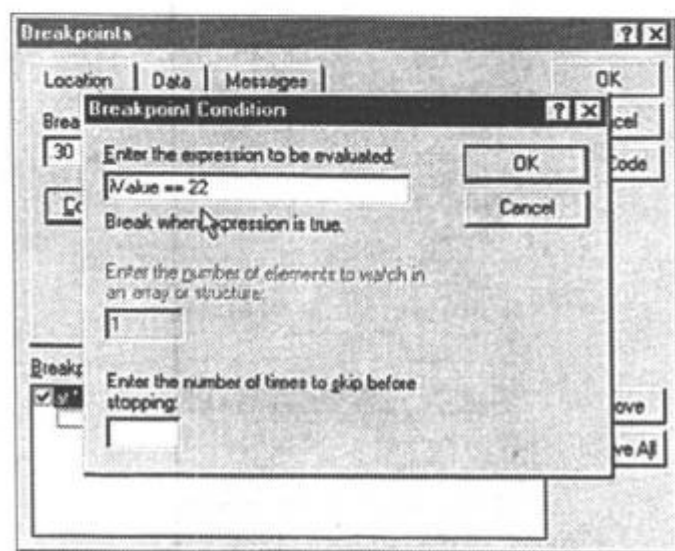


图 9-6 在第 30 行选择一个断点

图 9-7 设置断点条件为 `iValue==22`

从使用 Variable 窗口和 Watch 窗口(此处没显示)的常规调试中,可以发现传递到子程序中的 `iValue` 的值为 22。对于这一假想例子来说,该值是个“错误的数据”。

图 9-7 显示,设置的条件断点为,形参 `iValue` 等于 22。注意 Breakpoints Condition 窗口中表达式 `iValue==22` 下面的“Break when expression is true”。可以在表达式编辑框中输入任何计算值为 `true(!0)`或 `false(0)`的合法 C/C++语句。

通过输入跳过测试表达式的次数, Breakpoints Condition 窗口允许进一步完善条件断点。这可能是非常有用的,例如,讨论的这一子程序能够 5 次跳过 `iValues` 值为 22 的情况。第六个 22 就被解释为“错误数据”。

同其他情况一样,通过按 F5 键运行设置了断点的程序。图 9-8 显示 Debugger 已经停止。消息框详细说明了:断点语句所在的文件为 `brkcondts.cpp`,语句行数为 30, Debugger 停止的原因是 `iValue==22`。从前面的讨论中,可以知道,这时应该查看 Call Stack 窗口,检查是哪个子程序最后调用它,并传递了“错误数据”。这时,剩下的工作就是调试那个调用子程序。

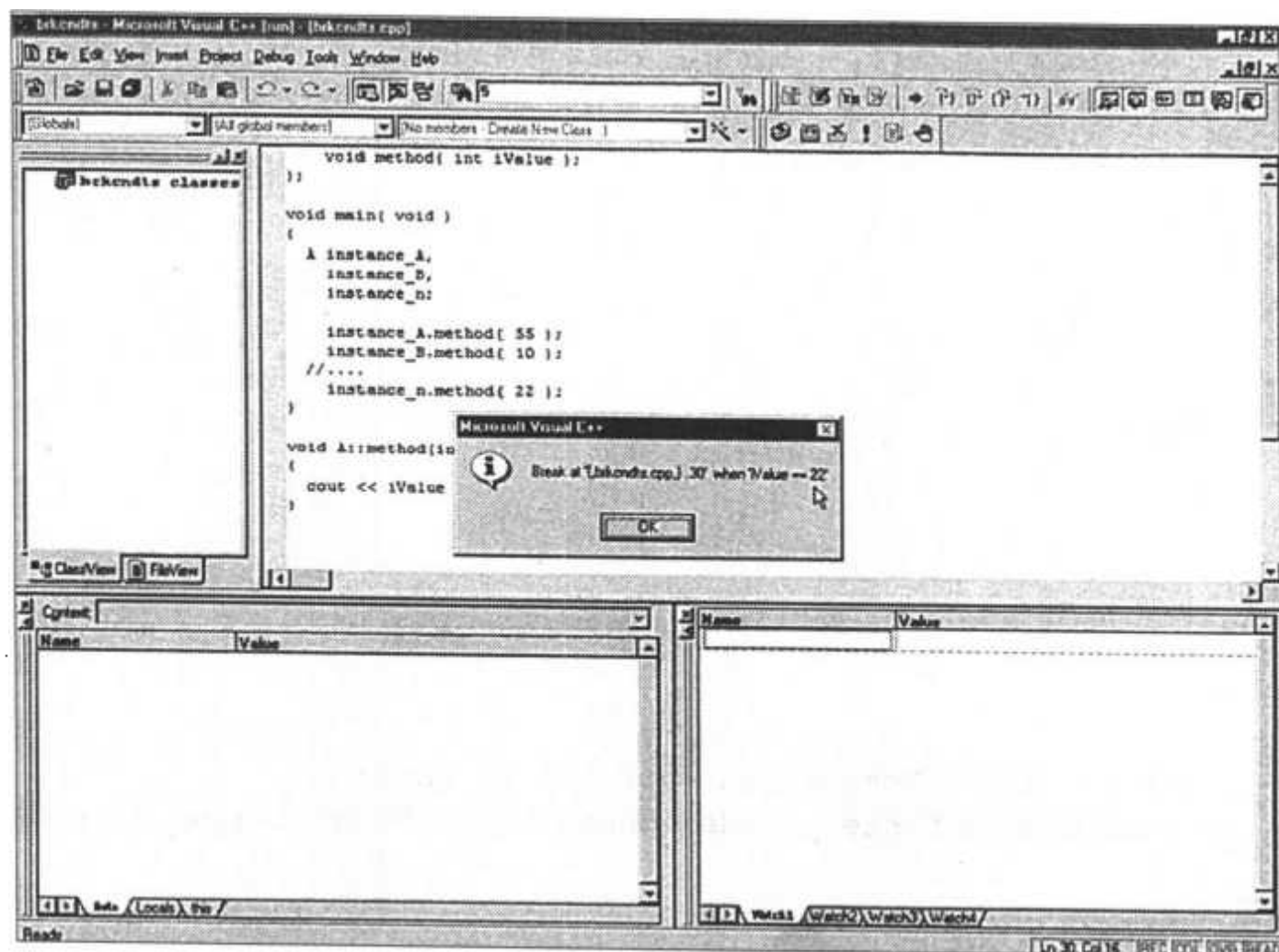


图 9-8 当 iValue=22 时 Debugger 停止

9.1.3 查找何处修改了指针

所有的程序员都知道，程序错误是我们学习语言基础的最好教材，C 和 C++也不例外。实际上，因为我们正在阅读这本书，所以已经知道 C/C++程序的错误是多么可恶，必须灵活掌握。例如，下面这一简单的面向对象的 C++程序(跳过解释段落，查看可否检测出错误)：

```
//
// badptr.cpp
// Detecting bad pointer assignment.
// Chris H. Pappas and William H. Murray, 2000
//
#include <iostream>
using namespace std;
typedef struct tagIntegerNode {
    int          iValue;
    tagIntegerNode *pNextIntegerNode;
} INTEGERNODE;
class A {
```



```
public:
    // made public to simplify tracing
    // would normally be protected:
    INTEGERNODE *pFirstIntegerNode;
};
class B:public A {
public:
    B( void );
};
class C:public A {
public:
    C( void );
};
void main( void )
{
    B instanceB;
    C instanceC;
    cout << instanceB.pFirstIntegerNode->iValue << endl;
    cout << instanceC.pFirstIntegerNode->pNextIntegerNode->iValue << endl;
};
B::B( void )
{
    pFirstIntegerNode          = new INTEGERNODE;
    pFirstIntegerNode->iValue    = 1;
    pFirstIntegerNode->pNextIntegerNode = NULL;
};
C::C( void )
{
    pFirstIntegerNode->pNextIntegerNode          = new INTEGERNODE;
    pFirstIntegerNode->pNextIntegerNode->iValue    = 2;
    pFirstIntegerNode->pNextIntegerNode->pNextIntegerNode = NULL;
};
```

这一短小的例子程序只是机械地实现了一种链表类型的算法。程序以定义链表节点 *INTEGERNODE* 开始，该节点具有一个整型数据成员 *iValue*，和一个指向下一个链表节点的指针 *pNextIntegerNode*。为了更有意义，父类 A 声明一个节点指针 *pFirstIntegerNode*，它将跟踪所有子类 B 和子类 C 定义的子节点。

同胞类 B 和 C 都仅有一个构造函数，其作用是动态地为一个 *INTEGERNODE* 节点分配内存，为其相应的指针赋值。对象 *instanceB* 的构造函数连接新节点地址到从父类 A 继承的 *pFirstIntegerNode* 数据成员上，而 *instanceC* 的构造函数连接新节点地址到节点对象 *instanceB* 的 *pNextIntegerNode* 数据成员上，因此我们创建了一个含有两个 *INTEGERNODE* 节点的简

单链表。

`main()`的任务除了实例化两个同胞类 B 和 C 外，还要输出赋给每个实例的值。从理论上讲，程序应当相应地输出整型值 1(`instanceB` 的 `iValue` 值)和 2(`instanceC` 的 `iValue` 值)。但是在 Debugger 中，执行该程序产生了如图 9-9 所示的结果。

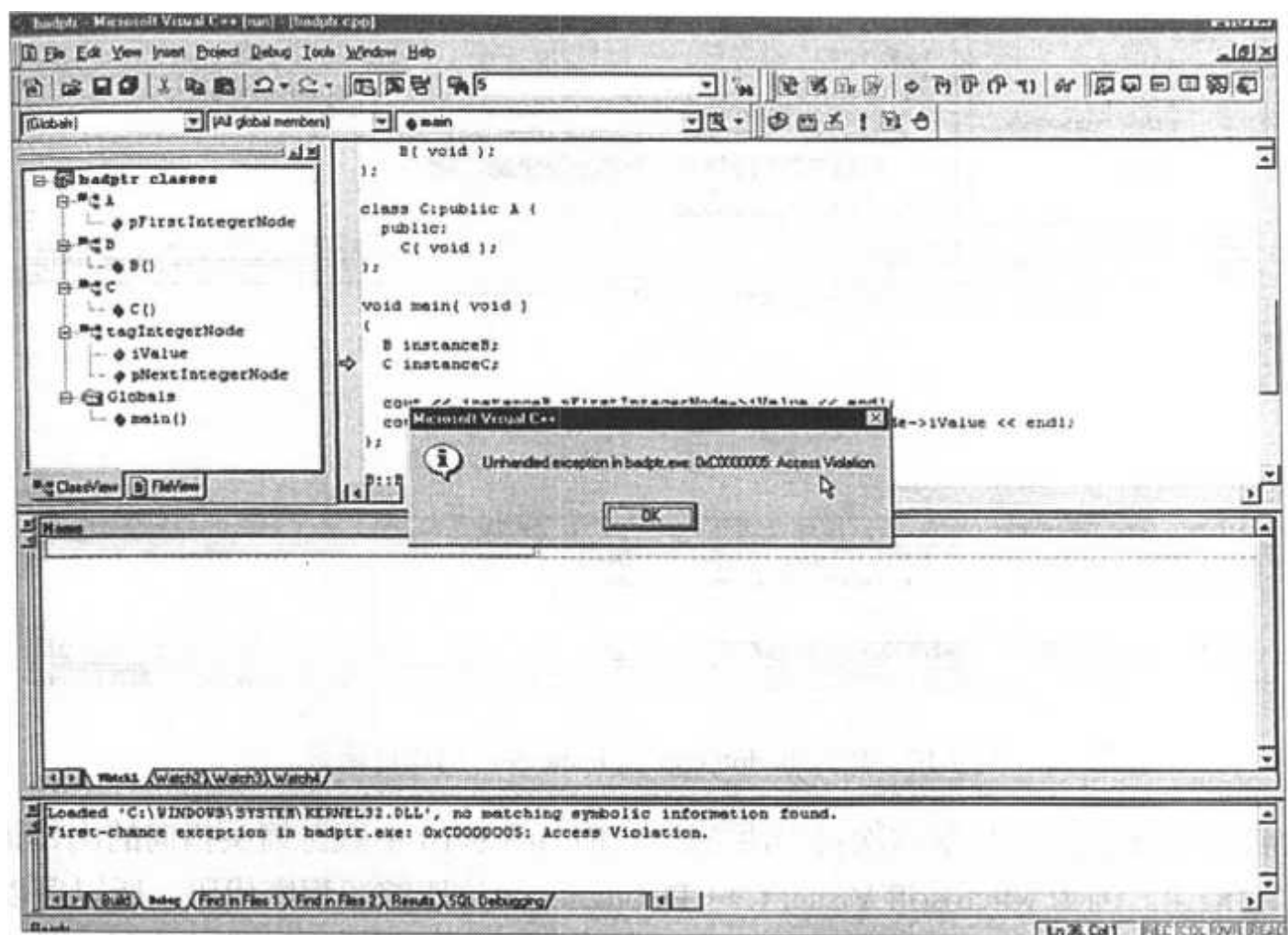


图 9-9 执行 badptr.cpp

仔细检查跟踪箭头的位置，表明是子类 C 的实例化产生了“Access Violation”（访问冲突）。

运用新掌握的调试技巧，我们决定重新启动 Debugger，并且不跳过该实例化语句（快捷键 F10），而是跳入（F11）`instanceC` 的构造函数，如图 9-10 所示。注意跟踪箭头现在停留在 C 的构造函数的第一个语句上。

再执行一步单步操作，“Access Violation”又出现了。Debugger 现在指向出错语句：调用 `new()` 函数并将新的 `INTEGERNODE` 节点赋给第一个节点的 `pNextIntegerNode` 指针。问题是，赋值语句的哪部分出错了？

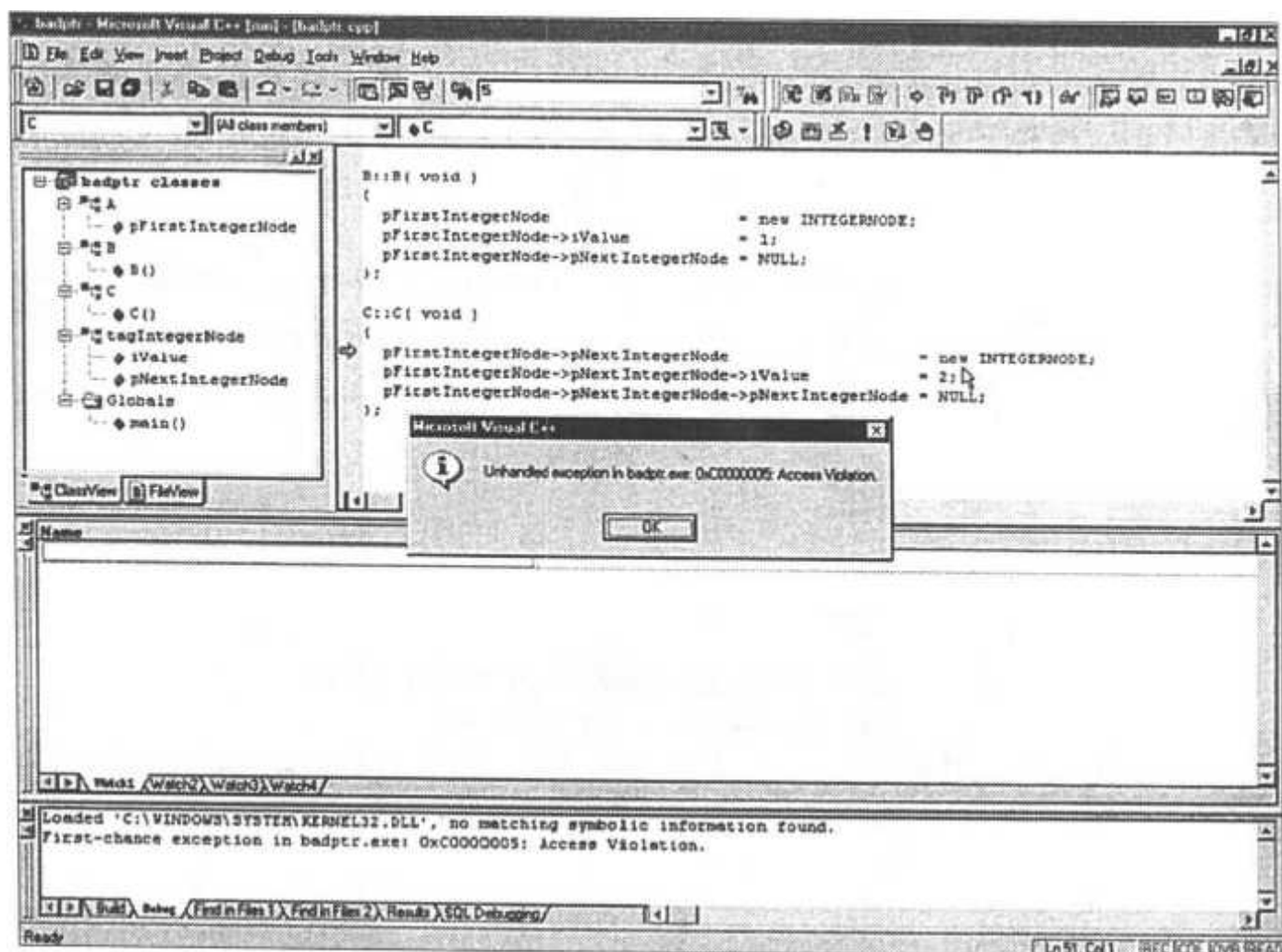


图 9-10 跟踪 badptr.cpp 到 instanceC 的构造函数

再次启动 Debugger，只是这次我们决定进入(F11)图 9-10 中跟踪箭头指向的出错语句，而不是跳过它(F10)。因为 Microsoft Visual C++ Debugger 允许跟踪源程序代码，所以可以由图 9-11 中跟踪箭头的位置看出函数 **new()**调用成功。之所以知道函数 **new()**成功执行，是由于如果这一函数本身就造成访问冲突的话，跟踪箭头不可能指到结尾括号处。然而，再按 F10 或 F11，离开对函数 **new()**的调用，将再一次产生原来在图 9-9 看到的“**Access Violation**”消息框。

结论是当试图将这一新的 **INTEGERNODE** 的地址赋值时产生了错误信息。现在问题变成了，如何应用 Debugger 找出是到底何处出了错？让我们一步一步地找到这一问题的答案。

首先，我们知道子类 B 的实例化执行正确，*InstanceB* 的构造函数看起来也执行了。那么在理论上，这就意味着 *pFirstIntegerNode* 一定已经产生了，并且被赋值为 *instanceB* 的新的 **INTEGERNODE** 对象的地址。于是我们决定将这二者均放在 Watch 窗口中。

然后，写下或记住 *instanceB* 新的 **INTEGERNODE** 对象的物理地址，查看 *pFirstIntegerNode* 中的地址是否变化过。

实际上，我们正在跟踪两个物理地址，第一个是编译器已经分配给指针变量 *pFirstIntegerNode* 的地址。这需要在变量名字的前面加地址操作符 **&**，将其以 **&pFirstIntegerNode** 的形式加到 Watch 窗口。

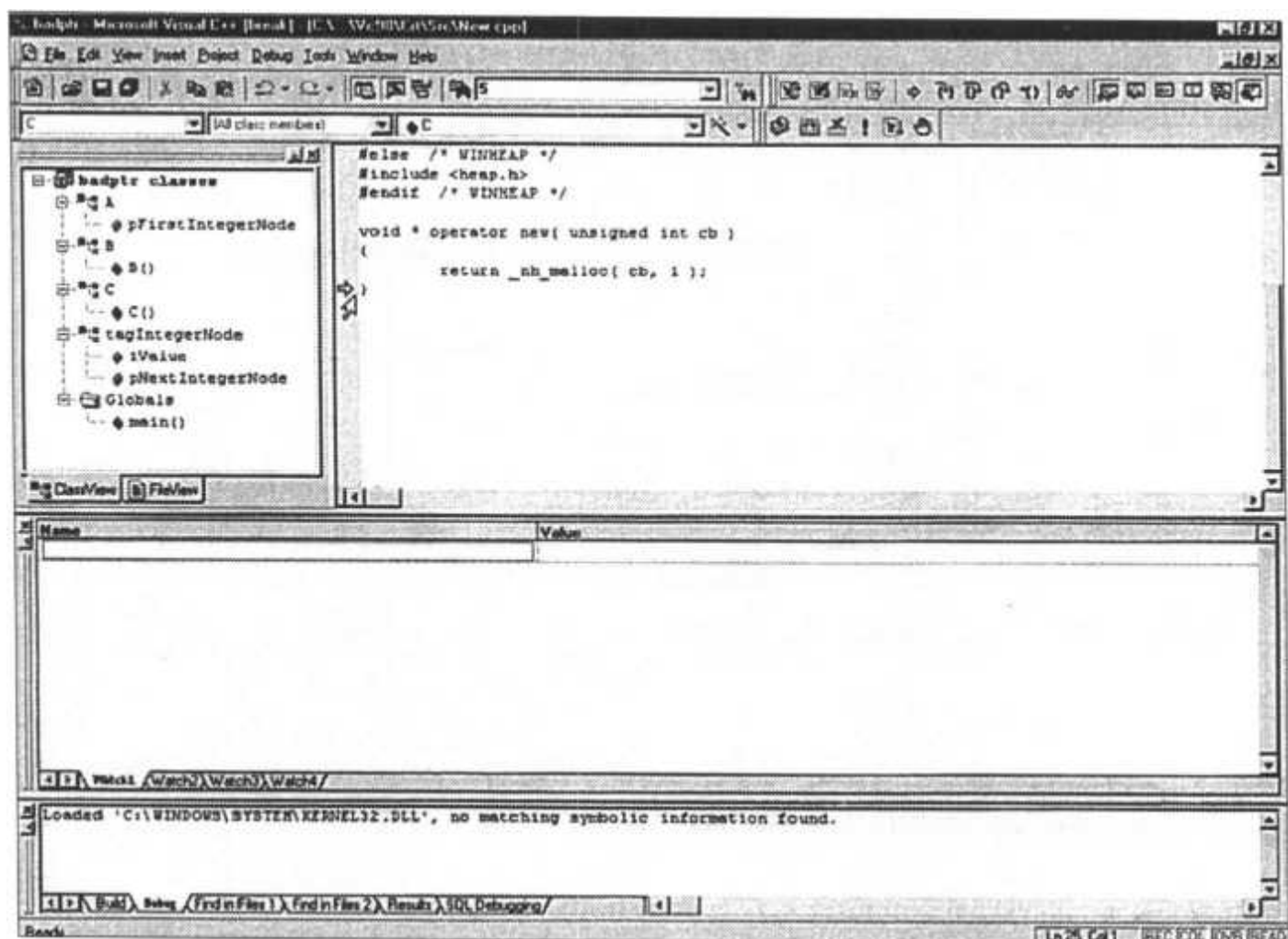


图 9-11 检测到 new() 函数成功执行

下面我们需要的证据是 *instanceB* 新的 *INTEGERNODE* 的地址。这是通过观察 *pFirstIntegerNode* (输入在观察窗口第二项, 如图 9-12 所示) 的内容完成的。这两个地址是 *0x0006afdf4* (&*pFirstIntegerNode*) 和 *0x007d0eb0* (指针变量 *pFirstIntegerNode* 中的内容)。

因为是如下的 *instanceC* 的语句:

```
pFirstIntegerNode->pNextIntegerNode=new INTEGERNODE;
```

引起的错误, 所以下面将试图检测 *pFirstIntegerNode* 的内容是否发生了变化。如果是这样, 有理由表明已经丢掉了 *instanceB* 的新的 *INTEGERNODE* 的正确地址。

这是 Microsoft 的 Visual C++ Debugger 真正闪光的地方。使用 Breakpoints 对话框中的 Advanced 选项, 可以指示 Debugger 在指针变量的指向地址内容变化时停止运行。

为设置这种类型的条件断点, 需要回到 Edit Breakpoints (ALT+F9) 菜单选项 (参见图 9-5 和图 9-6), 只是这次使用的不是 Breakpoints location 标签, 而是 Breakpoints Data 标签, 如图 9-13 所示。

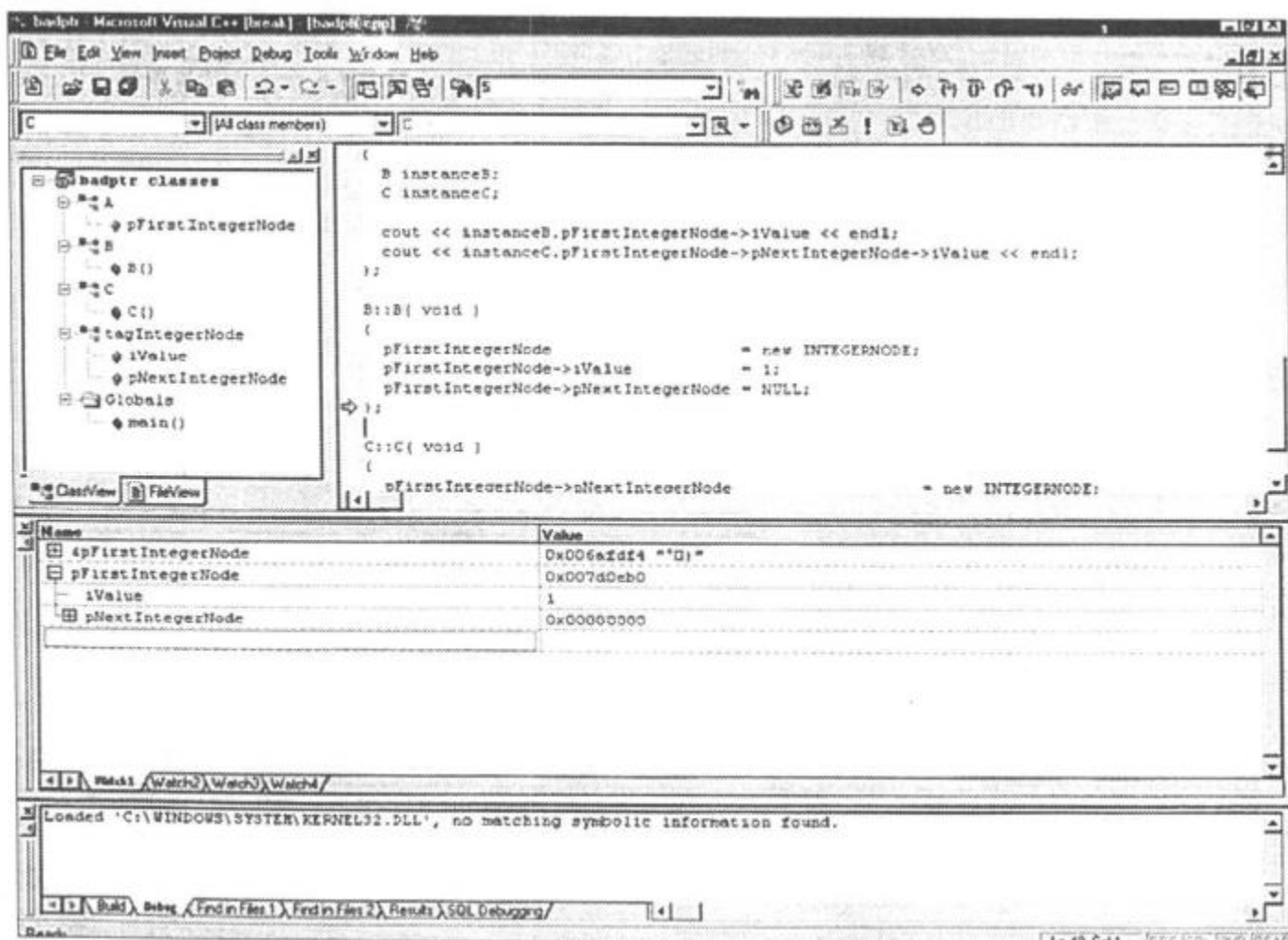


图 9-12 查看 `&pFirstIntegerNode` 物理地址和第一个有效的 `INTEGERNODE` 值

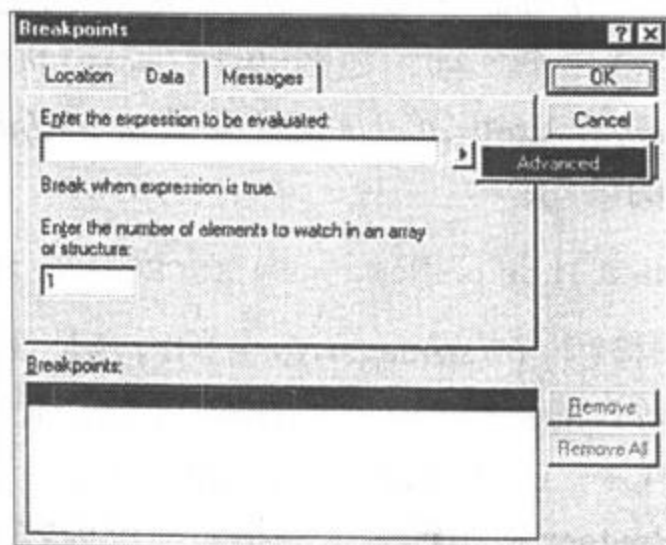


图 9-13 使用 Advanced Breakpoints 选项

因为涉及到父类和子类继承关系这样高度复杂的作用范围，所以需要使用 **Advanced Breakpoints** 选项(参见图 9-13)，单击 Enter the expression to be evaluated 编辑框右边的向右箭头激活该选项。如果不使用 **Advanced Breakpoints**，Debugger 是不可能标记指针变量 (`pFirstIntegerNode`)的地址变化，因为它没有能力处理具体实例的引用或作用域。

Advanced Breakpoints 窗口不仅需要输入一个表达式，还需要有与特定中断条件相关的函数名字、源文件名字和可执行文件名字(参见图 9-14)。这些完成以后，单击 OK 按钮，将看到一个与图 9-15 相似的窗口。

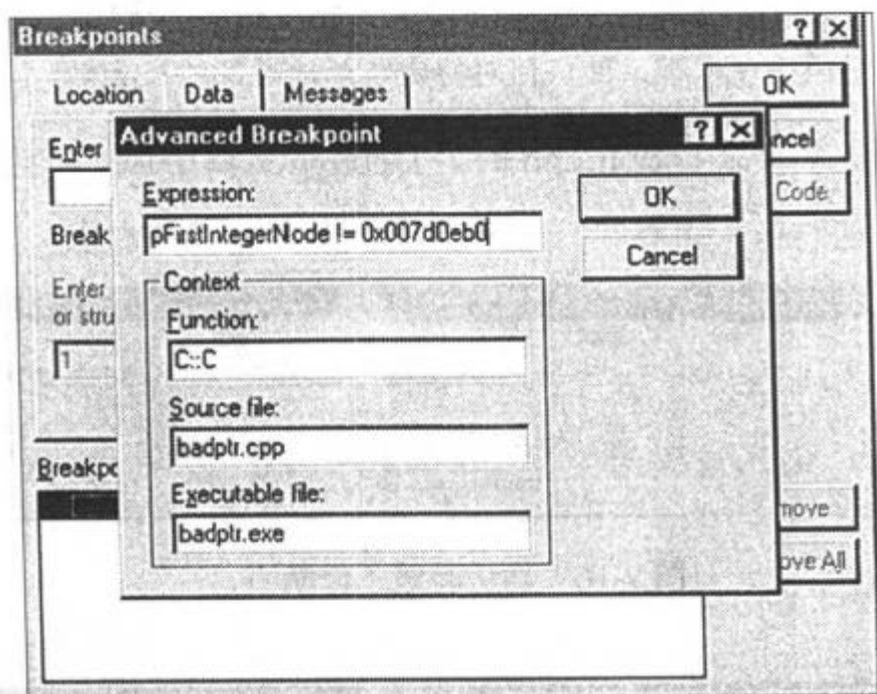


图 9-14 定义 Advanced Breakpoints

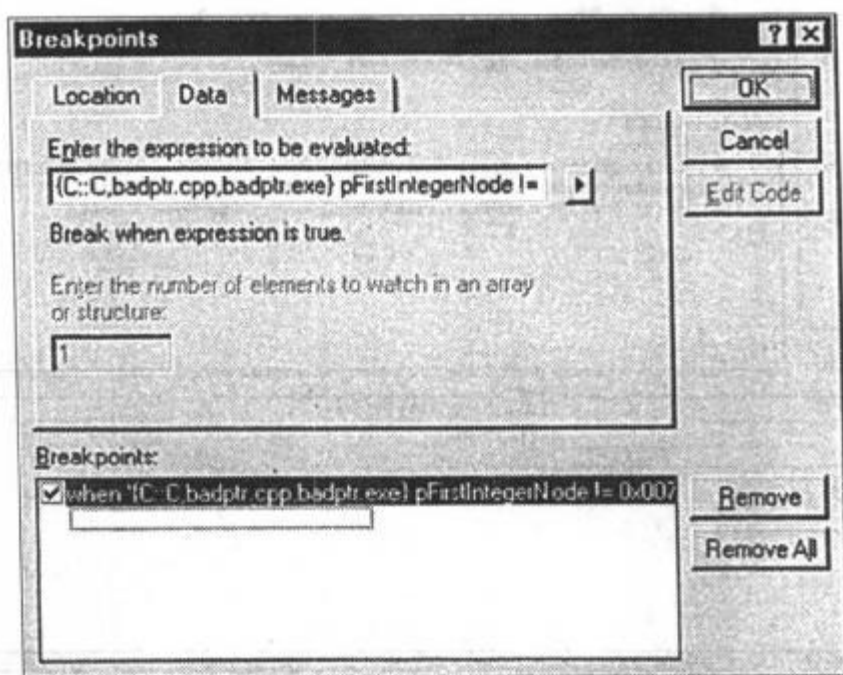


图 9-15 查看 Advanced Breakpoints 设置

在图 9-15 中，条件断点表达式 *pFirstIntegerNode != 0x007d0eb0* 为真时将暂停 Debugger，此时指针变量的地址将不再指向 *instanceB* 的新 *INTEGERNODE* 节点。如果算法正确，这种情况将不会发生。



断点设置如图 9-15, 现在要开始最后阶段的调试, 只需按 F5 键(运行到断点的热键)重新启动 Debugger 即可。

24x7

如果正在调试一个算法, 查找指定物理内存地址或物理地址指针的变化, 则不能增加或减少程序语句。否则将引起编译器重新分配对象物理内存, 使任何对特定物理地址的测试无效。

使用高级条件断点设置调试 badptr.cpp 时, Debugger 将中断并产生断点消息, 如图 9-16 所示。单击 OK 按钮, 显示跟踪箭头停在出错语句处(参见图 9-17)。

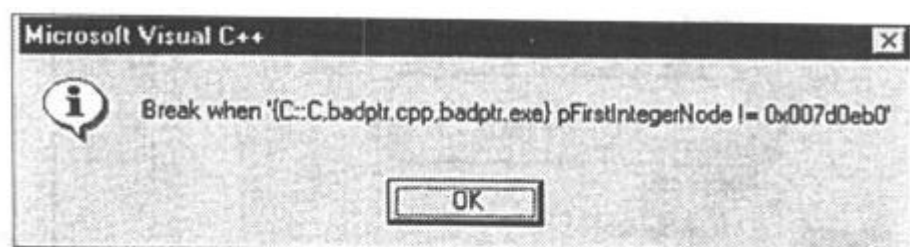


图 9-16 确认高级条件断点

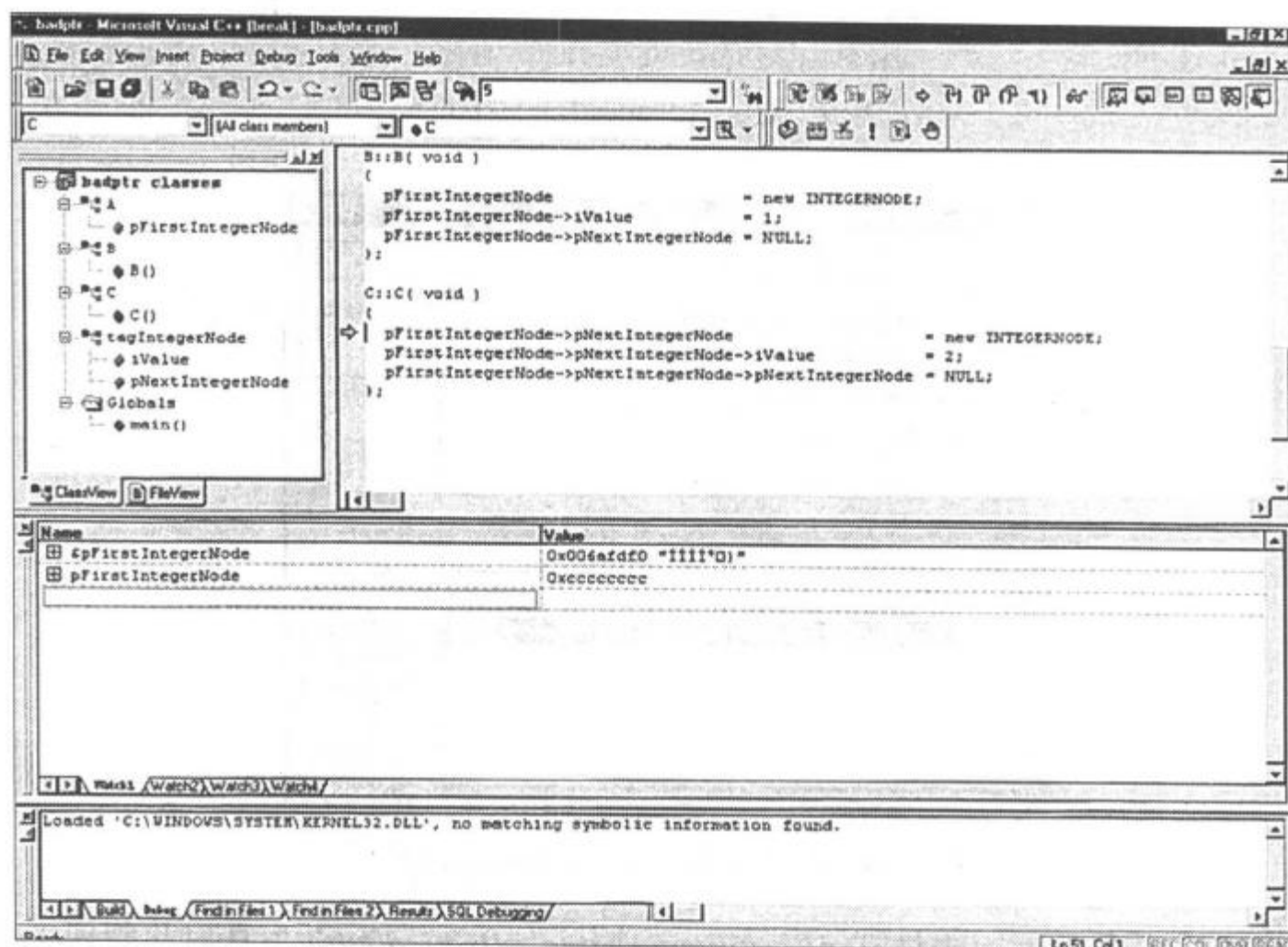


图 9-17 跟踪箭头指向出错语句

图 9-17 表明跟踪箭头停在引起“Access Violation”的同一行, 刚才产生了访问冲突的

跟踪掩盖了这条语句的出错之处。这一错误可能是对 `new()` 的调用，或赋值`=`，或对 `pNextIntegerNode` 的引用，还可能是指针 `pFirstIntegerNode` 本身引起的。

根据第一次跟踪，我们发现对函数 `new()` 的调用是成功的。接下来需要做的就是找出与那两个指针引用有关的任何内容。在 `Advanced Breakpoint` 的帮助下，我们现在知道在进入 `instanceC` 的构造函数时，`pFirstIntegerNode` 中存储的地址已经改变了——而它不应该改变！现在我们调试成功了，但是为什么程序不工作？

`badptr.cpp` 总是按其逻辑编译和装配，这是一个基本的面向对象设计方法的缺陷，涉及到指针、父类、子类的定义，以及继承关系。

问题在此处：父类 `A` 定义了指针 `pFirstIntegerNode`，`A` 被同胞类 `B` 和 `C` 继承，而这两者却没继承相同的 `pFirstIntegerNode`！

当子类 `B` 实例化时(`instanceB`)，其成员 `pFirstIntegerNode` 产生了一个具体实例内存单元，它被赋值为新的 `INTEGERNODE` 的地址。

随之而来的是同胞 `C` 的实例化(`instanceC`)。它也有对从 `A` 继承的 `pFirstIntegerNode` 的同一成员定义；然而，`instanceC` 分配的是其自己的 `pFirstIntegerNode`。由于 `instanceC` 的构造函数 `C::C` 使用这一实例特定的 `pFirstIntegerNode`，而这一 `pFirstIntegerNode` 没有被初始化，所以程序卡壳了。

如果不考虑花在调试算法上可能耗费的时间，修改工作将十分简单。那么如何修改？只需简单地加上 C/C++ 的关键字 `static`。请看下面改正后类 `A` 的定义：

```
class A {
public:
    //Made public to simplify tracing
    //Would normally be protected:
    static INTEGERNODE *pFirstIntegerNode;
};
```

该语法中的关键字 `static` 使任何子类定义不仅共享该成员的定义，而且通过地址共享了同一数据成员。

遗憾的是，在为类的数据成员添加 `static` 关键字时 C++ 有一个不那么灵活的语法要求。C++ 要求 `static` 数据成员在外部声明——也就是在正式类定义的外边，比如：

```
//Static data members must be initialized at file scope, even
//if private.
INTEGERNODE *A::pFirstIntegerNode;
```

下面的程序 `staticptr.cpp`，用来帮助说明如何修改代码以适应整个算法(加粗语句说明了最低限度的修改代码)：



```
//
// staticptr.cpp
// Detecting bad pointer assignment
// Chris H. Pappas and William H. Murray, 2000
//
#include <iostream>
using namespace std;
typedef struct tagIntegerNode {
    int          iValue;
    tagIntegerNode *pNextIntegerNode;
} INTEGERNODE;
class A {
public:
    // Made public to simplify tracing
    // Would normally be protected:
    static INTEGERNODE *pFirstIntegerNode;
};
// Static data members must be initialized at file scope, even
// if private.
INTEGERNODE *A::pFirstIntegerNode;
class B:public A {
public:
    B( void );
};
class C:public A {
public:
    C( void );
};
void main( void )
{
    B instanceB;
    C instanceC;
    cout << instanceB.pFirstIntegerNode->iValue << endl;
    cout << instanceC.pFirstIntegerNode->pNextIntegerNode->iValue << endl;
};
B::B( void )
{
    pFirstIntegerNode          = new INTEGERNODE;
    pFirstIntegerNode->iValue    = 1;
    pFirstIntegerNode->pNextIntegerNode = NULL;
};
C::C( void )
{
```

```
pFirstIntegerNode->pNextIntegerNode      = new INTEGERNODE;
pFirstIntegerNode->pNextIntegerNode->iValue  = 2;
pFirstIntegerNode->pNextIntegerNode->pNextIntegerNode = NULL;
};
```

修改代码后的 `staticptr.cpp`，其算法将按照预期执行，分别输出整型数值 1 和 2。`badptr.cpp` 和 `staticptr.cpp` 最重要的一点是，使用通常可以使用的 Debugger 工具，有技巧地一步步调试了这些简单的面向对象的算法。

另外，在 `badptr.cpp` 中所发现错误的特殊之处在于，它不仅必须使用条件断点，而且要用高级断点。通过这一新的 Debugger 工具，现在我们知道如何克服简单断点表达式所固有的作用域问题，这种表达式没有能力告诉 Debugger 如何跟踪指定的实例成员。

9.2 ClassView 窗口要素

当运行 Microsoft Visual C++ Studio 时，最左边的窗口是 Workspace。它可以分成两个独立的窗口，ClassView 窗口和 FileView 窗口。ClassView 窗口直观地展示了类数据成员和类成员函数或方法之间的关系。

下面的程序定义了一个典型的父类和子类，二者都有 **public**、**private** 和 **protected** 类型的数据成员和方法。每一个成员，即数据和函数都有与其作用相对应的特定名字。由于 ClassView 窗口中成员符号非常小，所以成员的命名应当可以形象地反映每个成员在程序中的作用。

```
//
// clasview.cpp
// Understanding ClassView symbols.
//
// external static definition
// required for static class data members
static int static_publicParentData;
class parent {
public:
    int      publicParentData;
    static int static_publicParentData;
    parent() {}

private:
    float    privateParentData;
    void     privateParentMethod() {};

protected:
    double   protectedParentData;
    void     protectedParentMethod() {};
};
```



```
class child:public parent {
public:
    int        publicChildData;
    void        publicChildMethod() {};
    child() {};

private:
    float        privateChildData;
    void        privateChildMethod() {};

protected:
    double        protectedChildData;
    void        protectedChildMethod() {};
};

inline void doNothingFunction( void ) {};

void main( void )
{
    parent instanceParent;
    child instanceChild;
}
```

图 9-18 给出了 *clasview.cpp* 的一个典型创建, 其中最大化了 Workspace 窗口和 Edit 窗口, 以便于查看。



图 9-18 clasview.cpp 的 ClassView 窗口

图 9-19 是 Workspace 窗格中的 ClassView 窗口的一个特写镜头。用加粗字体表示的工

程名字，即 `clasview classes`，代表了缺省的工程结构。展开此工程时，ClassView 窗口显示工程中包含的类。如果展开某一个类，将显示这一类中的成员。

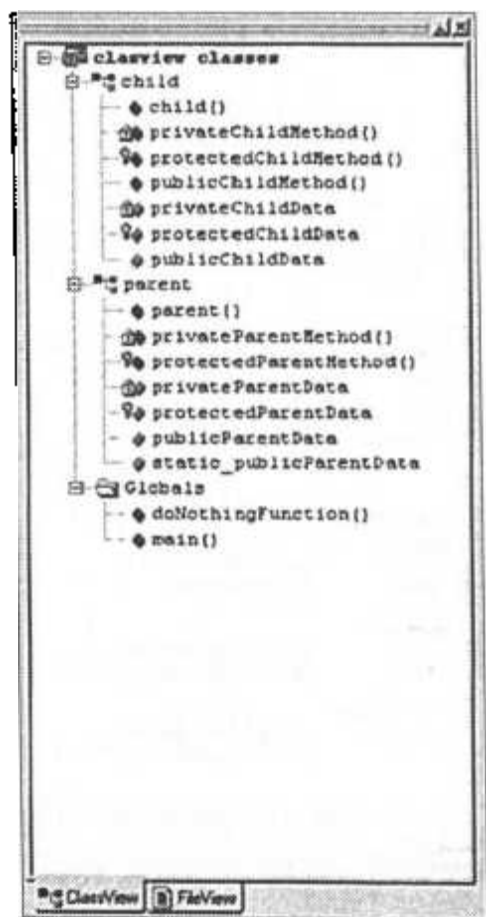


图 9-19 Workspace 的 ClassView 窗口的特写镜头

如果使用标准的 Windows 颜色，图上近乎黑色的符号在屏幕上为深蓝色的，灰色的符号在屏幕上为紫色的，而浅灰色的符号在屏幕上为浅蓝色的。

父类或子类的成员函数(方法)是以紫色表示的。父类或子类的数据成员是以浅蓝色表示的。成员函数的符号是一个向左斜的长方体，数据成员的符号是一个向右斜的长方体。

ClassView 窗口中的图标传达了工程中类和类成员的附加信息。当类成员数据或方法是全局可访问时，其符号不作任何装饰。**Protected:**数据成员或成员函数被装饰了一个小钥匙图标，**Private:**数据成员或成员函数被装饰了一个带钥匙孔的锁。这些装饰物直观地提示了类的成员的作用范围，即类相对程序、子类相对父类的关系。

9.2.1 ClassView 窗口的 Grouped by Access 功能

可以通过在类定义上右击，重新排列视窗中的图标的顺序。图 9-20 选择了 `parent` 类，并采用了 *Group by Access* 显示格式。

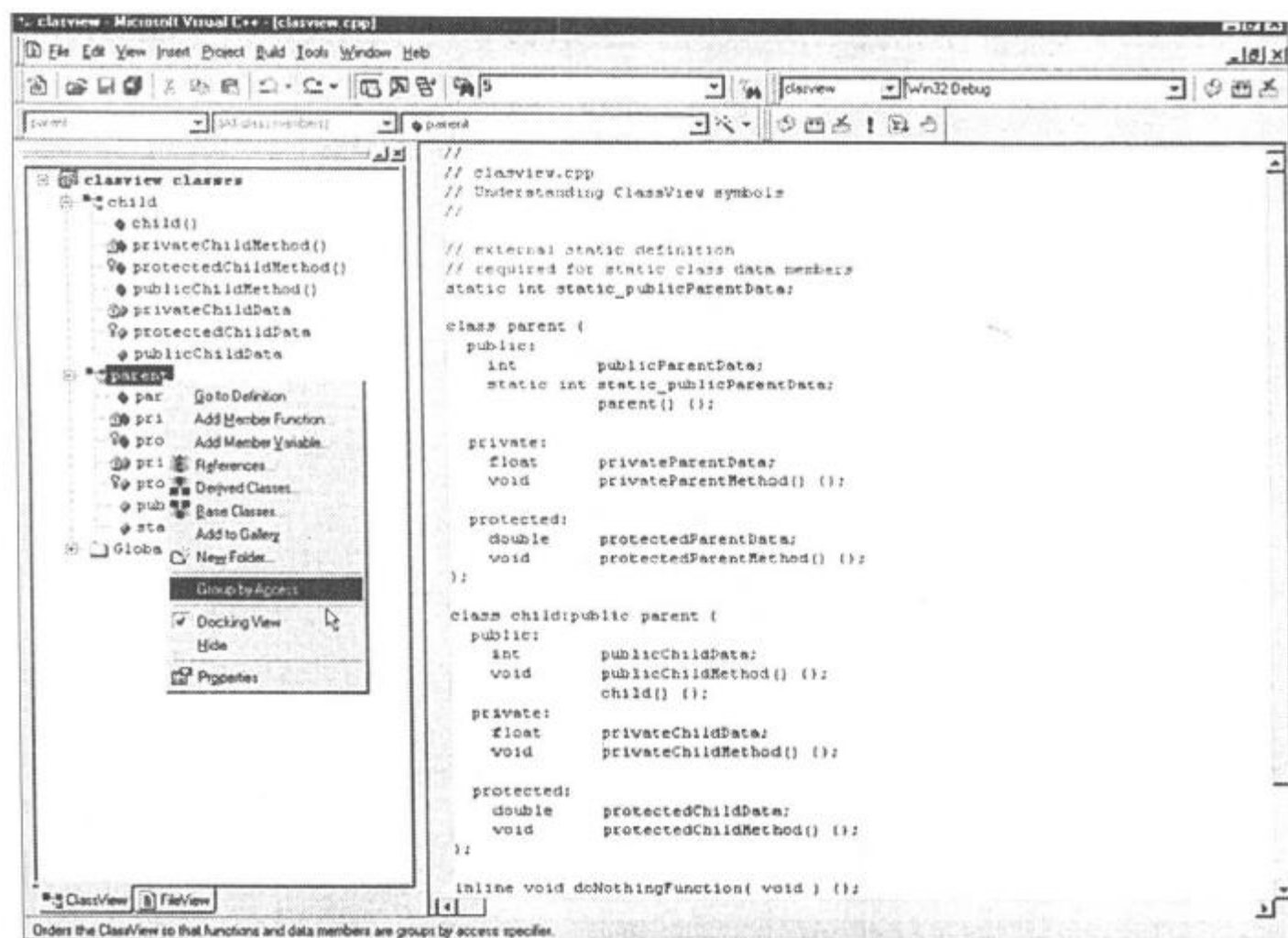


图 9-20 使用 Group by Access 选项对 ClassView 窗口的内容排序

图 9-21 显示的是按 *Group by Access* 重新排序后的 ClassView 窗口。

这对于观察对象是非常有用的，因为它按从最低级保护 **public:**到最高级保护 **private:**的顺序排列类成员。

9.2.2 ClassView 窗口的 Base Classes 功能

图 9-22 显示使用 ClassView 窗口中的本地菜单(右击 *parent* 类)以详细说明 *clasview.cpp* 的基类。

选择本地菜单的 Base Classes 选项，打开 Base Classes and Members 窗口，类似于图 9-23 所示的情况。f、d 和 S 代表类的函数(function)、数据(data)和静态定义(Static definition)。

从 Base Classes and Members 窗口可以找到所有要了解的与应用程序基类有关的内容。该窗口显示了所有的基类(左边的窗格)，也显示了所有的 **public:**、**private:**和 **protected:**类型的成员，包含类定义的源文件名和路径，以及引用基类的源文件。双击源文件名，Debugger 将自动将该文件在编辑窗口中打开。下面还有更详细的介绍。

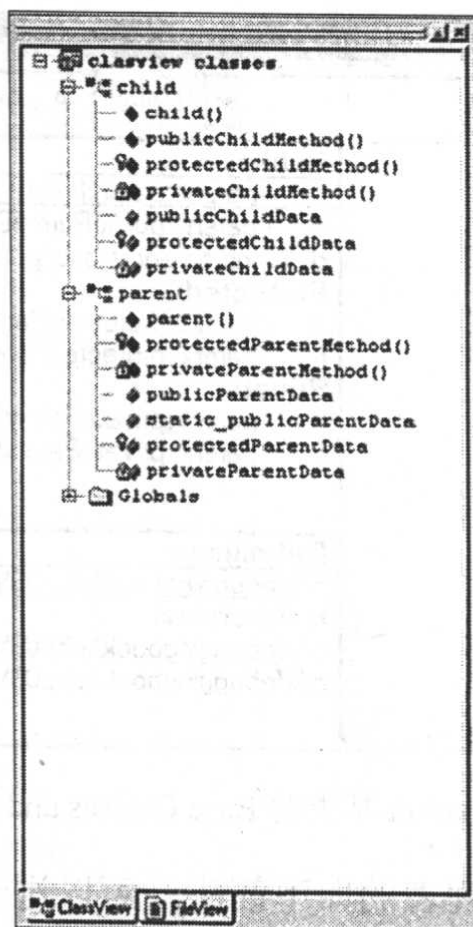


图 9-21 执行 Group by Access 后的 ClassView 窗口

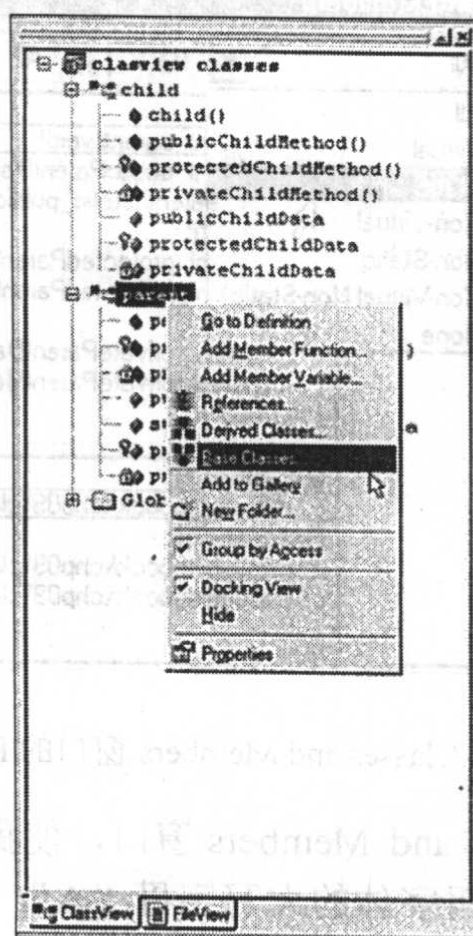


图 9-22 找出有关 clasview.cpp 所有基类的详细资料

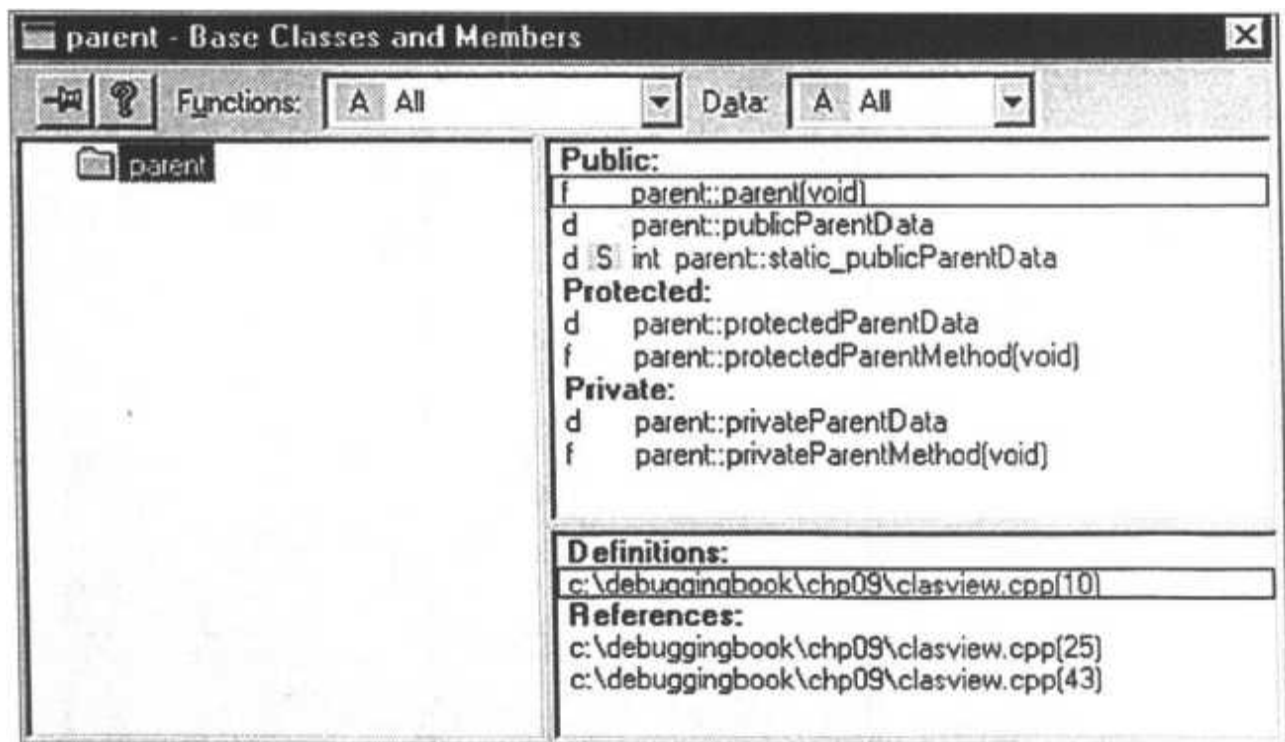


图 9-23 显示 parent 基类的 Base Classes and Members 窗口

单击 Functions 或 Data 下拉列表的向下箭头，可以进一步提高查看效率。图 9-24 显示了 Functions 下拉列表的选项。

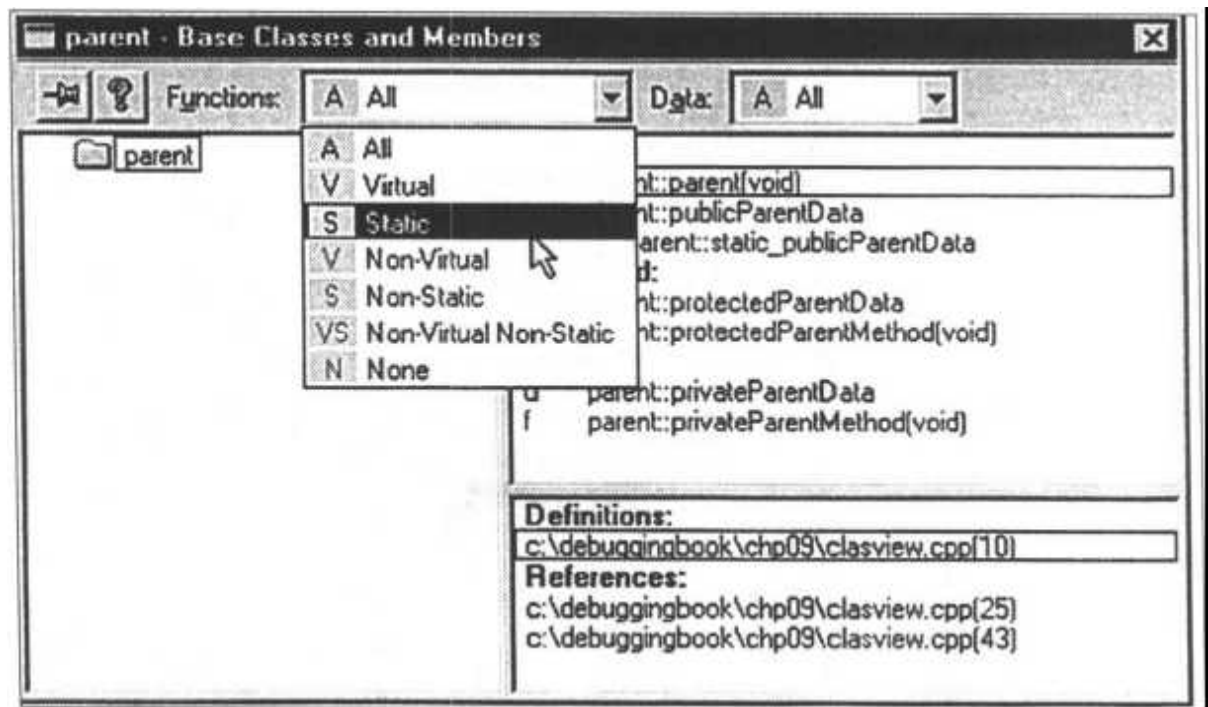


图 9-24 选择 Base Classes and Members 窗口的 Functions 下拉列表

该列表允许调整 Base Classes and Members 窗口，使我们只看到静态、虚、非虚、非静态、或非虚非静态成员函数。加粗字体的大写字母“A”、“V”和“S”都有对应的非粗体字母。粗体表示活动限定符；非粗体表示非活动限定符。

单击 Data 下拉列表，如图 9-25 所示，显示类似的数据查看选项。

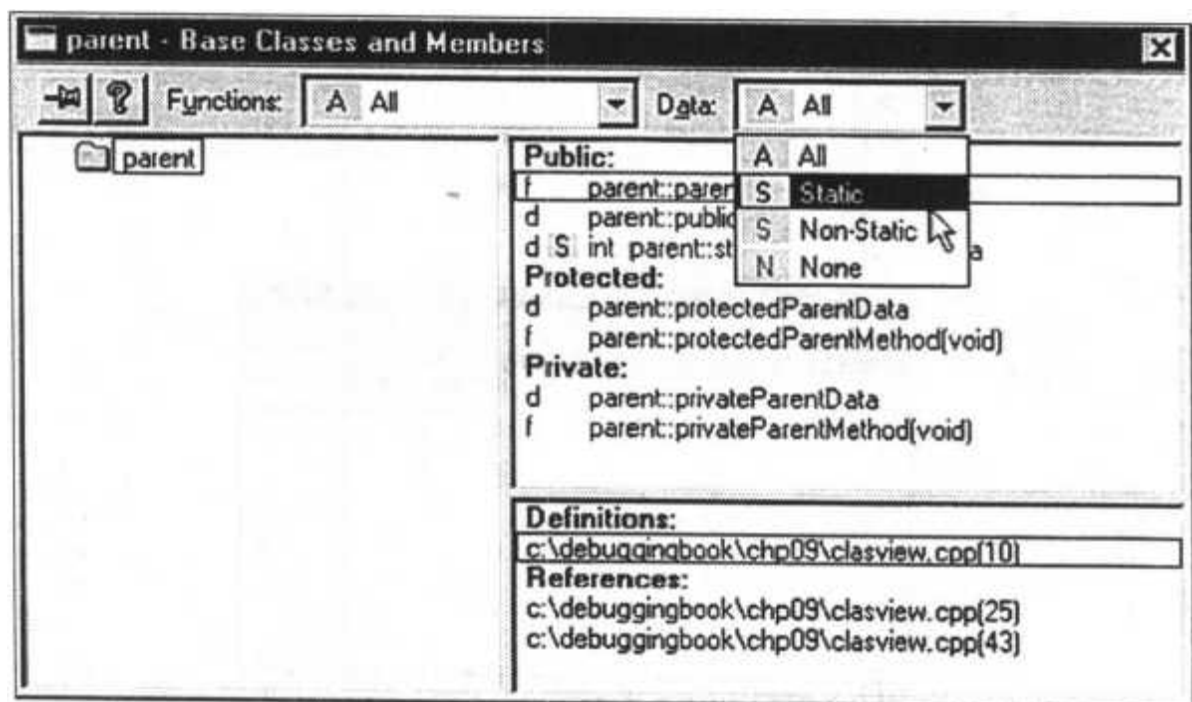


图 9-25 选择 Base Classes and Members 窗口的 Data 下拉列表

静态数据选项对于仔细检查任何用于生成链表算法的父类、基类或根类是理想的。请记住这些选项的位置和激活方法。切记，掌握 Debugger 要花时间，其功能强大无比，但面向对象程序产生的错误也不得了！

9.2.3 ClassView 窗口的 References 功能

如果感兴趣的只是确定哪些文件引用了特定对象，那么无须使用 Base Classes and Members 选项。图 9-26 显示的是针对 *child* 类选择了 ClassView 窗口的本地菜单(见前面图 9-22)中 Reference 选项的情况。

Definitions and References 窗口的左边窗格列出了所选对象的名字，右边窗格列出了使用所选对象的所有文件。双击某个文件名，将自动在编辑窗口中打开该源文件。

9.2.4 ClassView 窗口的 Derived Classes 功能

最后一个要介绍的 ClassView 窗口本地菜单的选项是 Derived Classes(见前面图 9-22)，它允许查看父类与其子类的继承关系。图 9-27 显示这一选项针对 *clasview.cpp* 的 *parent* 类被激活的情况。

初步观察表明家族树在 Derived Classes and Members 窗口的左边窗格中。双击任何子类(在本例中只有一个 *child* 类)将刷新窗口的内容，显示出派生类的信息，如图 9-28 所示。

假如这一 *child* 类被用作其他子类的父类，那么可以重复前面的步骤，向下嵌套窗口的内容直到最后的子类。

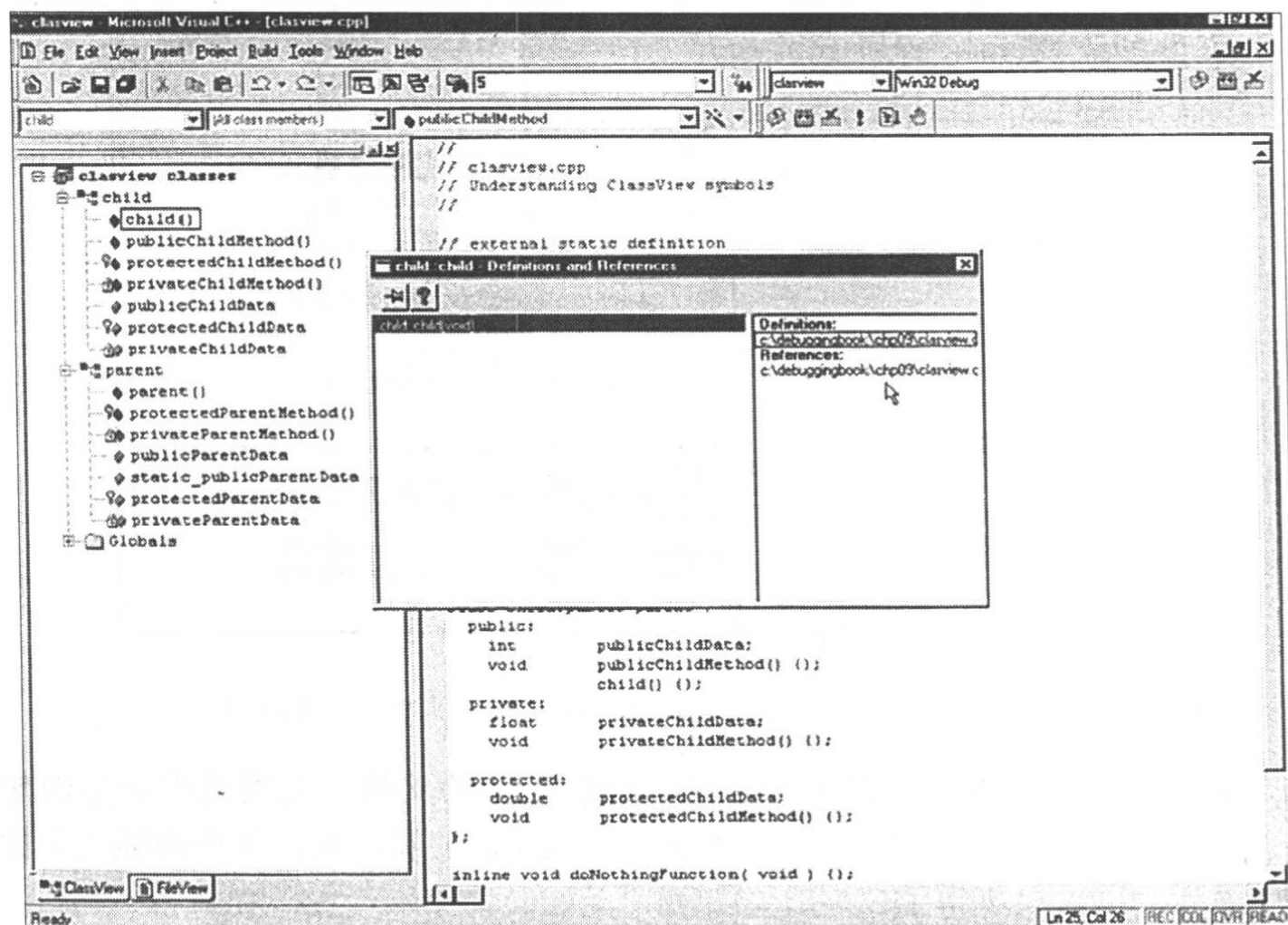


图 9-26 child 类的 References 窗口

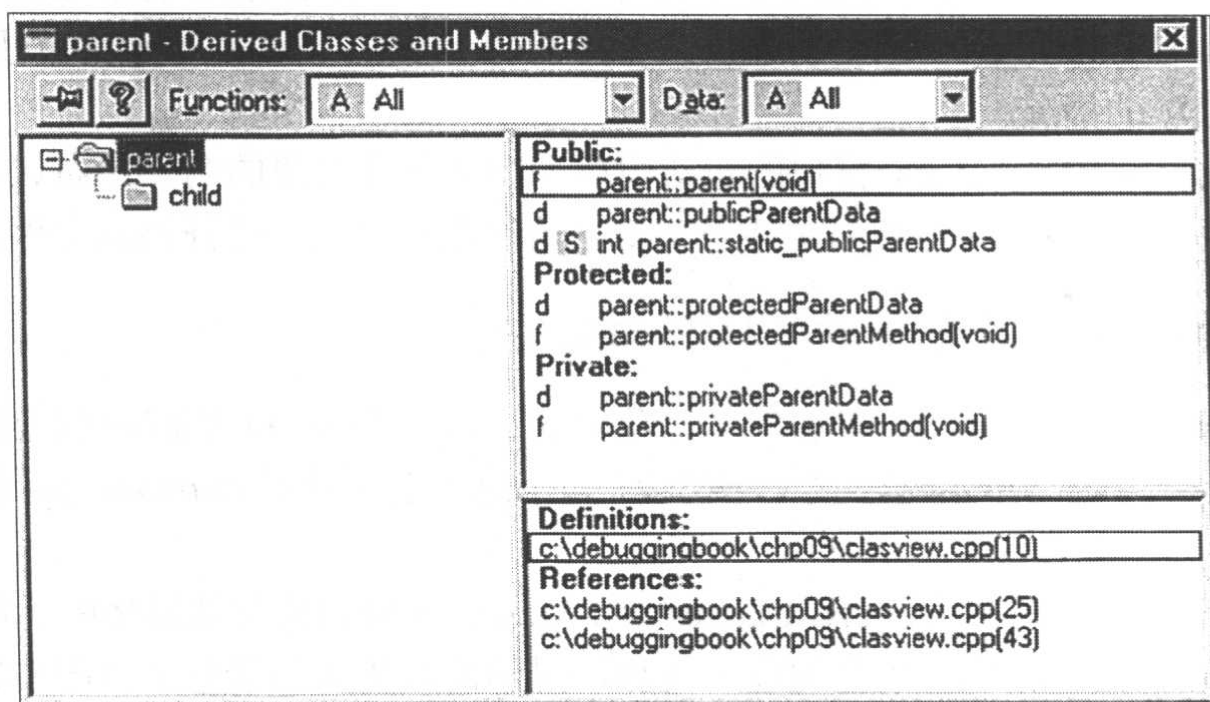
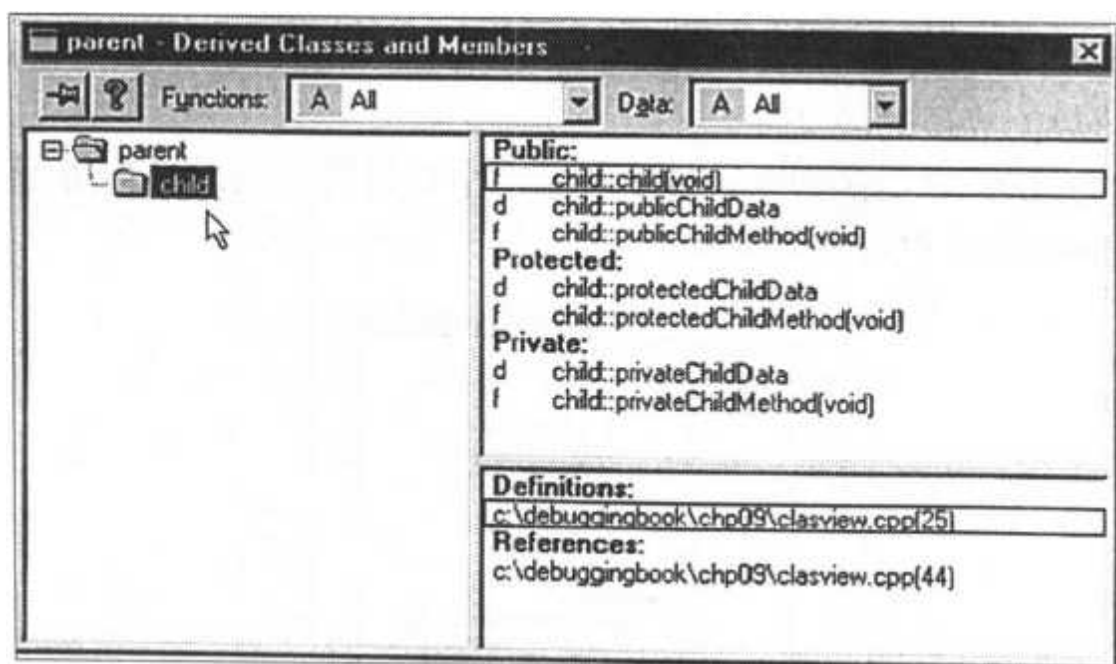


图 9-27 使用 Derived Classes and Members 窗口查看 clasview.cpp 的 parent 类


 图 9-28 使用 Derived Classes and Members 窗口查看 *clasview.cpp* 的 *child* 子类

9.2.5 ClassView 窗口中菜单的其余项

调试面向对象的算法时，可发现 ClassView 窗口的本地菜单中还有两个选项很有用。右击 ClassView 窗口中的一个类数据成员，出现了第一个选项。图 9-29 显示了针对 *child* 类的成员 *protectedChildData* 选择 ClassView 窗口的本地菜单的情况。

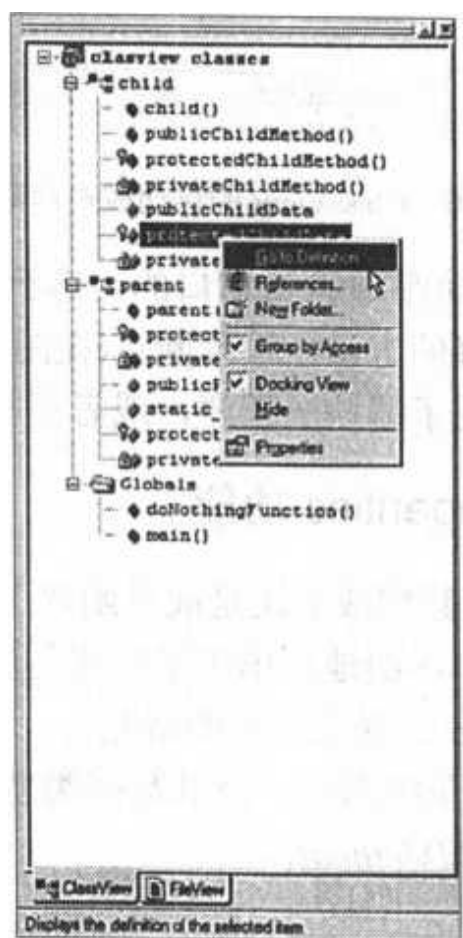


图 9-29 ClassView 的浮动数据成员菜单



注意，可以使用本地菜单中的特定选项自动跟踪编辑窗口到定义数据成员的文件处，或列出所有引用此数据成员的源文件。

右击一个类的方法或成员函数名，将发现第二个有用的选项。图 9-30 显示了针对 *parent* 类的构造函数 *parent()* 选择此选项的情况。

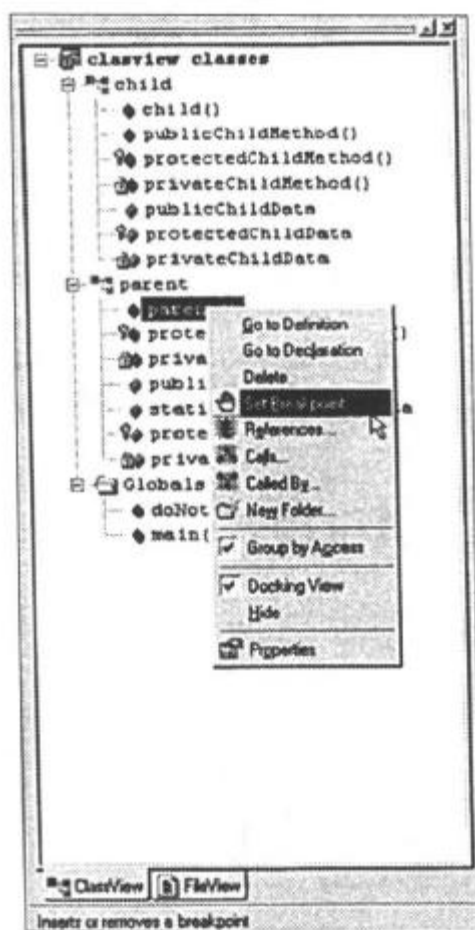


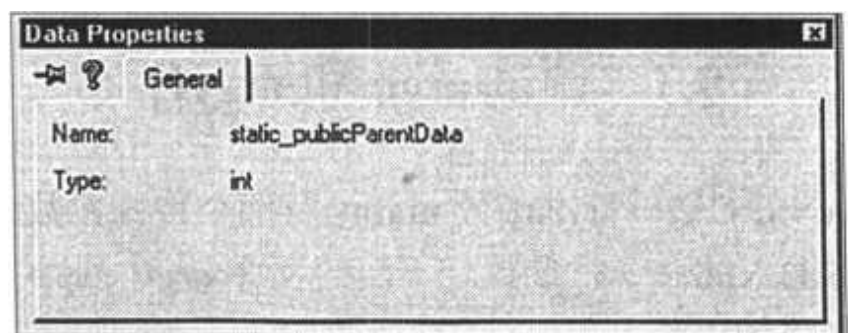
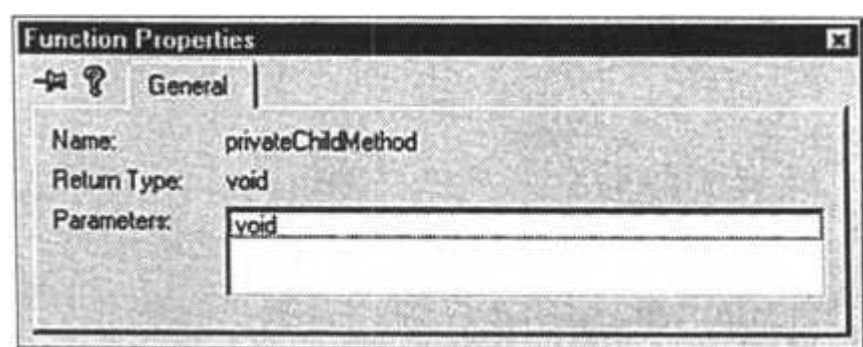
图 9-30 ClassView 的浮动成员函数菜单

此处有意义的选项包括，自动跟踪编辑窗口的内容到定义该方法的源文件处，以及任何引用此方法的文件处，或在选择的方法上设置断点(在图 9-30 中的反白选项)。我们甚至可以使用这些菜单找出所选方法调用了哪些子程序，或者哪些子程序调用了所选方法。

9.2.6 ClassView 窗口的 Properties 功能

在 ClassView 窗口中，无论在数据成员还是成员函数上右击，弹出的菜单都有 Properties 选项。选择这一选项，打开的窗口详细地列出了指定成员的名字和数据类型。图 9-31 显示了针对成员 *static_publicParentData* 选择此选项的情况。

Data Properties 窗口显示了变量的名字，及其数据类型格式。图 9-32 选择同样的菜单选项，只是这次针对的是 *privateChildMethod()*。


 图 9-31 *static_publicParentData* 的 Data Properties 窗口

 图 9-32 *privateChildMethod* 的 Function Properties 窗口

Function Properties 窗口显示了子程序的名字及其返回值类型和形式参数表。

9.2.7 在 ClassView 窗口中添加文件夹

通过在 ClassView 窗口中添加新的文件夹，可以方便地组织工程中的类。新的文件夹可以包含属于该工程的任何类，但不能包含如文本文件或资源文件这样的非类文件。为组织其他类型的文件，到其相关窗口，例如，FileView 窗口或 ResourceView 窗口。

为在 ClassView 窗口中添加文件夹，首先移动鼠标指针到要添加文件夹的工程处。然后右击，选择本地菜单中的 New Folder 选项。可以在文件夹中嵌套文件夹，但不能在一个类中嵌套文件夹。

9.2.8 在文件夹之间移动类

一旦为工程生成一个新的文件夹，即可简单地将任何需要的类拖进该文件夹以组织工程中的类。注意不能将类从一个工程中移入另一工程的文件夹中。

9.2.9 隐藏或显示 ClassView 窗口

平常工作时可能只需要打开几个窗口，可以定制工作平台隐藏一个或几个窗口。为隐藏或显示 ClassView 窗口，将鼠标指针指向 Project Workspace 下边的某一标签，右击。在查看列表中，选中或清除 ClassView 项锁定其显示方式。



9.3 调试 argc 和 argv[]

许多程序员不知道 ANSI C/C++ 标准函数 `main()` 有两个特殊参数 `argc` 和 `argv[]`。这两个参数获取程序开始执行时输入的命令行参数。下面的程序 `argc_v.cpp` 演示了在面向对象环境中使用 `argc` 和 `argv[]` 所必需的语法。

```
//  
// argc_v.cpp  
// Debugging command-line arguments.  
//  
#include <cmath>  
#include <cstdlib>  
#include <iostream>  
using namespace std;  
class sample {  
public:  
    sample(int argc, char **argv);  
};  
sample::sample( int argc, char **argv )  
{  
    int iAccumulator = 0;  
    if( argc == 1 ) {  
        cout << "You only enetered the program name.\n"  
              << "The program will now exit.";  
        exit( EXIT_FAILURE );  
    }  
    for( int offset = 1; offset < argc; offset++) {  
        cout << *++argv << endl;    // or argv[ offset ];  
        iAccumulator += atoi( *argv );// or argv[ offset ];  
    };  
    cout << iAccumulator << endl;  
}  
void main( int argc, char *argv[] )  
{  
    sample ARGV_V( argc, argv );  
}
```

我们所感兴趣的第一个语句是 `main()` 函数的定义语句:

```
void main( int argc ,char *argv[ ] )
```

`argc` 包含命令行输入参数的个数, 而 `argv[]` 是一个指向实际参数的一维字符型指针数

组，字符指针是一个以 ‘\0’ (null) 结尾的字符串。例如，要运行程序 `argc_v.cpp`，可以输入下面的命令：

```
c:\argc_v 1 2 3 4 5 6
```

`argc_v` 是程序的名字，`1 2 3 4 5 6` 是程序的参数，都在命令行输入。要调试一个带参数 `argc` 和 `argv[]` 的程序，必须从命令行运行来测试其算法。这就带来一个有趣的问题：即，如何从 Visual C++ Studio 外部启动程序，而同时继续留在 Visual C++ 的 Debugger 中？解决的方法要使用到 Project|Settings 窗口中的一个少有人知的选项。

图 9-33 显示了调试 `argc` 和 `argv[]` 所必需的开始步骤。首先，选择菜单 Project|Settings (ALT+F7)。

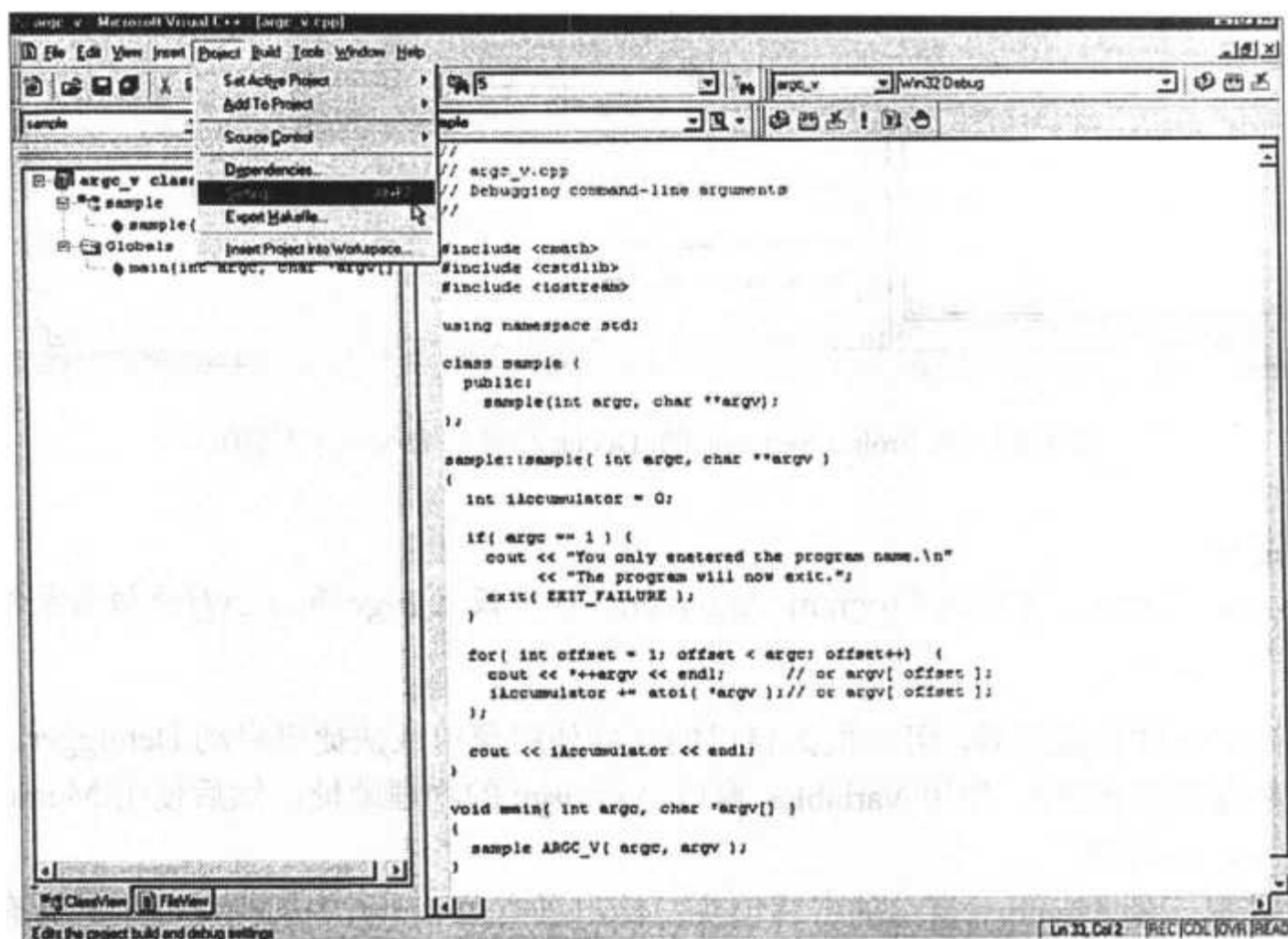


图 9-33 为 `argc` 和 `argv[]` 设置 Debugger

在 Project Settings 窗口中，单击 Debug 标签。此处我们所关心的是标题为 “Program arguments” 的框。此处可放置测试参数值或初始条件，我们在的例子中，参数是整型数 `1 2 3 4 5 6` (参见图 9-34)。

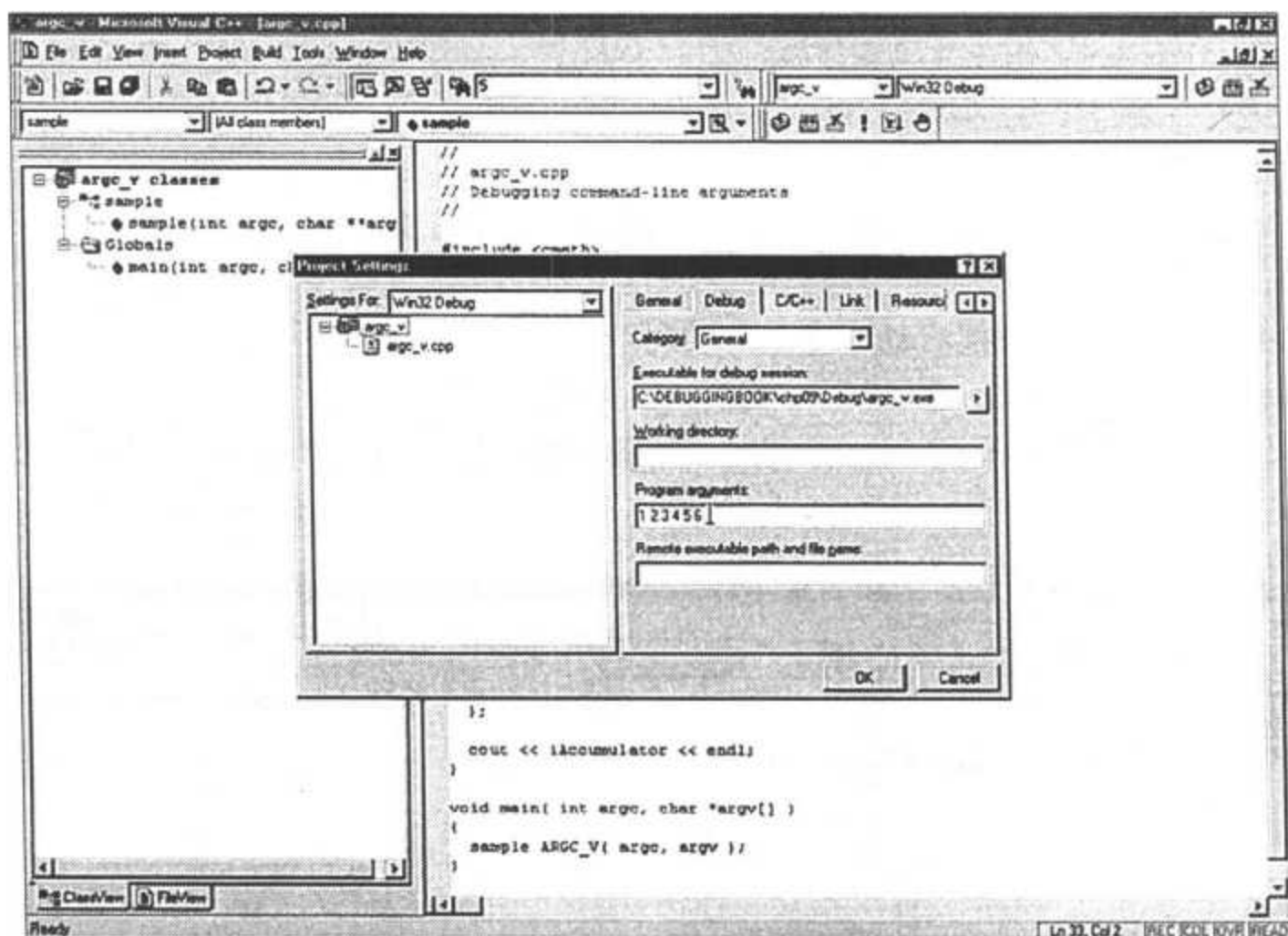


图 9-34 在 Project Settings 的 Debug 标签中输入 argv[] 的值

错误监视

在 Project Settings 窗口的 Program Arguments 框中设置 *argc* 和 *argv[]* 的初始条件时，不必输入程序的文件名。

输入初始条件参数值后，可以根据自己的喜好使用菜单或快捷键启动 Debugger。图 9-35 显示了按 F10(跳过)键后，使用 Variables 窗口显示 *argv* 的物理地址，然后使用 Memory 窗口自动跟踪 *argv* 的内容。

图 9-36 显示的 Memory 窗口的特写镜头，给出了有关 *argv[]* 中的指针引用的所有信息。

要理解这一程序，需要知道 *argc* 被初始化为 7。*argc* 中为字符串的计数值，字符串包括程序名，即 *argv[0]*，然后是字符表示的整型数值 1 2 3 4 5 6(所有 *argv[]* 指向的条目都保存为以 '\0' 为结尾的字符串)，保存在 *argv[1]* 到 *argv[7]* 中。

为了便于讨论，程序 *argc_v.cpp* 使用了两种语法引用 *argv* 自身。例如，第二个参数在 *main()* 中表为 *char *argv[]*，而在类 *sample* 的构造函数原型中表示为 *char **argv*。

二者都在逻辑上表示同一种数据类型；只是第二种形式隐藏了 *argv* 实际上是一个字符型指针数组这一事实。

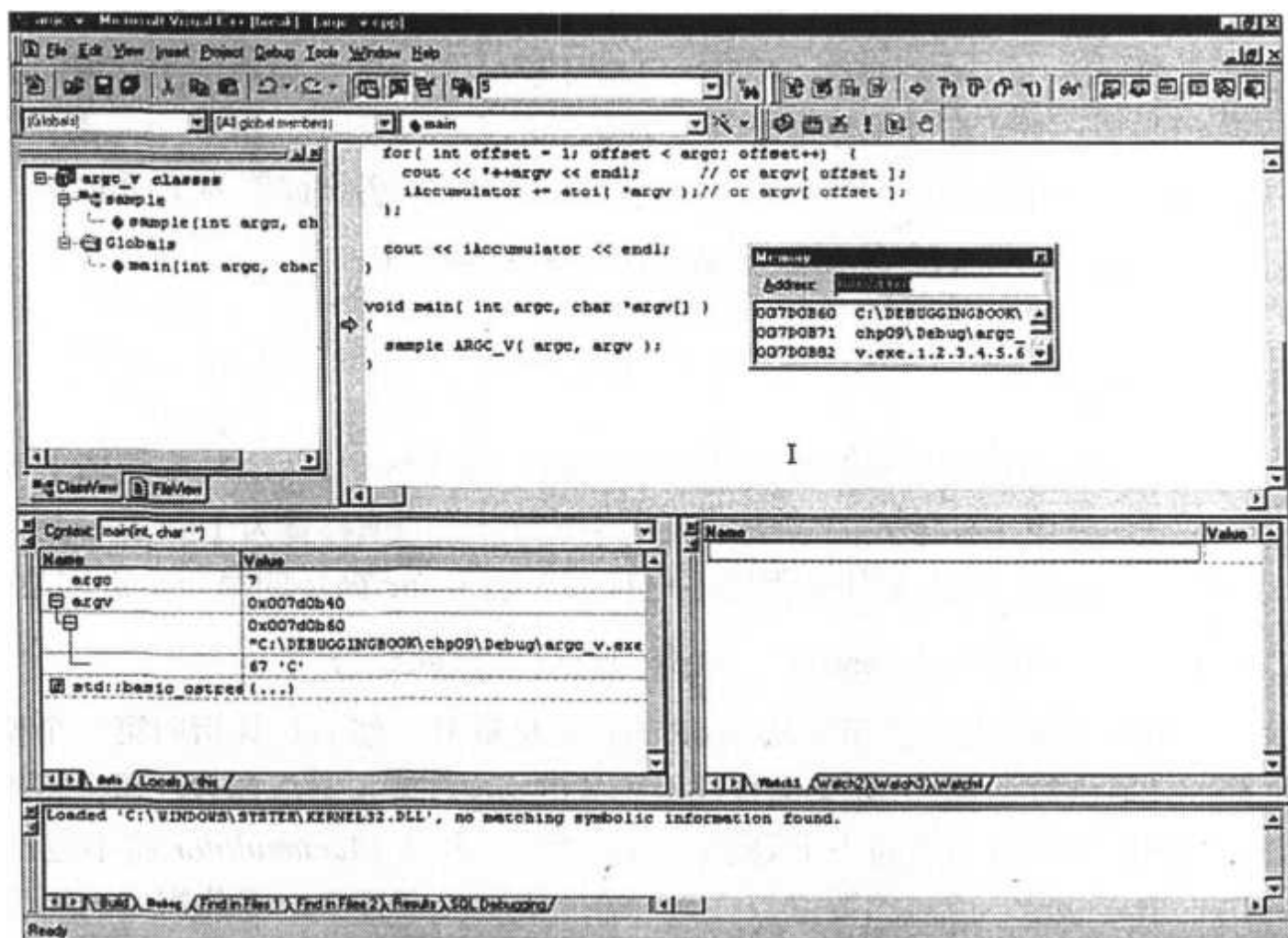


图 9-35 使用 *argv[]* 的 Variables 窗口的地址打开 Memory 窗口

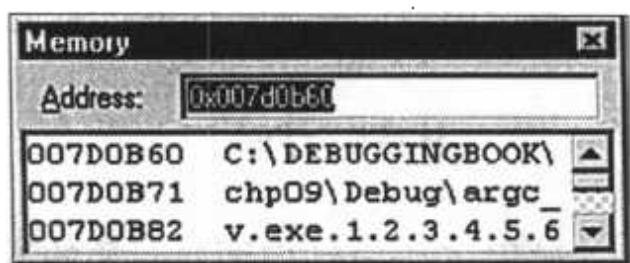


图 9-36 显示 *argv[]* 内容的 Memory 窗口的特写镜头

构造函数 *sample::sample* 以测试 *argv* 的值开始，这一值如果是 1，意味着用户运行此程序不带任何整型参数。因为缺少有效的整型数据，构造函数算法的其余部分将变得无意义，所以程序执行 *exit()*。

```
if( argc ==1) {
    cout << "You only entered the program name.\n"
        << "The program will now exit.";
    exit( EXIT_FAILURE );
}
```



for 循环说明了使用 *argv* 值的两种形式的语法：指针即 **++argv* 和 **argv*，以及另一种选择：数组形式即 *argv[offset]*。注意，用来访问指针数组 *argv* 的条件不同于常规的 **for** 循环。大多数处理数组的 **for** 循环都以偏移地址为 0 开始。然而在本例中，第 0 个元素保存指向程序名字符串的指针，而不是第一个整型数值。这就是为什么 *offset* 被置为 1 而非 0。

```
for( int offset = 1; offset < argc; offset++) {  
    cout << *++argv << endl;    //or argv[ offset ];  
    iAccumulator += atoi( *argv );//or argv[ offset ];  
}
```

最后，大多数 **for** 循环使用检验条件 *offset < MAX*。在本例中，MAX 在逻辑上等于 *argc* 的计数值。如果算法使用下标形式的语法，那么将得到合法的偏移量为 1 到 7——换句话说，检验条件为 *offset <= argc*。那么为什么程序不这样写？因为 **for** 循环的第一个语句是：

```
cout << *++argv << endl;    //or argv[ offset ];
```

前置递增操作 **++argv* 首先把指针从 *argv* 的首地址移开，然后让其指向第一个整型数值即 1，再执行指针取内容运算 ***。如果选择数组形式的 *argv* 即 *argv[offset]*，那么将不得不将 **for** 循环的检验条件由小于(<)改为小于等于(<=)。否则，和值 *iAccumulator* 将不包括 6。

最后，由于所有 *argv* 指向的数值被存为 ‘\0’ 结尾的字符串，因此需要调用合适的转换程序将其变成数值型的。例子中的程序调用了字符型到整型的转换函数 *atoi()*：

```
iAccumulator += atoi( *argv ); //or argv[ offset ];
```

Microsoft Visual C++ 有五个转换函数，即 *atof()*、*atoi()*、*_atoi64()*、*atol()* 和 *atold()*。

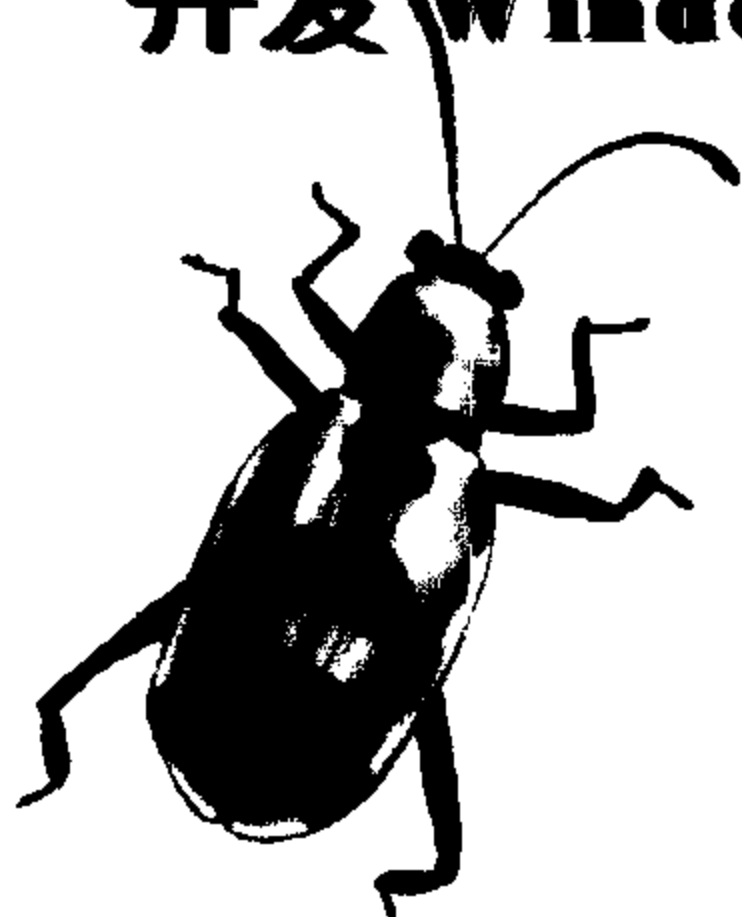
9.4 小结

当程序的解决方法由直观的过程 C/C++ 转向面向对象的程序设计时，不可避免地增加了代码的复杂性。不幸的是，同样，调试阶段的复杂性也增加了。在本章中，我们介绍了 Visual C++ 如何用附加的调试命令、特性和前面讨论的 Debugger 功能的高级用法来帮助我们解决问题。

在下一章，将学习如何用 Debugger 处理 MFC 编码的算法。MFC 是 Microsoft 的标准 Windows 对象的再组合，设计它是为了尽可能快地建立和运行 Windows 应用程序。 ■

第 10 章

使用 MFC 类库
开发 Windows 程序





绝大多数程序员都是从面向过程 Windows 程序设计开始起步的，就如同本书第 8 章中提到的那样。随着工程规模的扩大和应用程序中资源数量的增多，面向过程平台的缺点变得越来越明显了。过去有一个时期，Windows 程序员们曾经高兴地待在这两个阵营中。现在，对于一那些在工程中包含了 DLL、ActiveX、COM，甚至还有集成了 STL 的重要开发程序，我们怀疑是否还用面向过程平台开发。

对于刚才提到和其他尚未提到的应用程序类型，我们需要具有可重用能力的类，也就是 Microsoft Foundation Class(MFC)类库。封装了的 MFC 是强有力的开发和调试工具，它包括 Application Wizard(应用程序向导)和 Class Wizard(类向导)。MFC 类库封装了所有常用的面向过程的窗口函数，其对控制条、属性页、OLEO、ActiveX 控件和许多其他功能提供支持。此外，对于数据库开发，提供了包括 DAO 和 ODBC 等众多数据源的支持。

本章将作为 MFC 程序开发的入门读物，重点在于使用各种 VC++ 工具开发面向对象 Windows 程序的重要概念。我们假设读者在面向对象设计方面有很好的基础，能理解第 9 章中介绍的内容，并熟悉 MFC。对于从来没有用 MFC 编过程序的读者，Chris H. Papas 和 William H. Murray 编写的《*Visual C++: The Complete Reference*》一书(Osborne/McGraw-Hill 出版公司出版)可以当作快速入门书。

10.1 为什么使用类库

MFC 类库提供了程序员容易使用的对象。甚至在诸如 C 这样非面向对象语言结构中，面向过程的 Windows 程序开发也追求面向对象程序设计中好的思想。C++ 和 Windows 程序设计结合是自然的，这能充分利用面向对象的特性。MFC 类库提供了对 Windows Application Program Interface(API)的广泛实现。MFC 类库在一组可重用类中封装了最重要的数据结构和 API 函数。

MFC 类库提供了比 C 程序员使用的传统函数库更多的方便。

设计提示

将 MFC 类库代码加入 Windows 工程包括：

- 消除了函数名和变量名的冲突。
- 封装了类中的代码和数据。
- 增加了继承性。
- 因为有设计良好的类库，所以减小了代码长度。
- 得到的类表现为语言的自然扩充。

在使用 MFC 类库的基础程序中，创建一个窗口需要的代码能够从 100 行减少到 30 行。

绝大多数类库设计为类的层次结构，由父类派生子类，子类再继续派生下级子类。在

下节，我们将分析 MFC 类库的基础。

10.2 一个真正的基础类——CObject

CObject 是 MFC 类库中的一个重要的父类，其广泛地使用在 Windows 应用程序开发中。MFC 类库的头文件，通常都放在 Mfc\Include 子目录中，其中包含了定义 MFC 类的大量信息。

让我们简要地查看定义在 afx.h 头文件中的 CObject 类的一个片段：

```

////////////////////////////////////
// class CObject is the root of all compliant objects
class CObject
{
public:
// Object model (types, destruction, allocation)
virtual CRuntimeClass* GetRuntimeClass() const;
virtual ~CObject(); // virtual destructors are necessary
// Diagnostic allocations
void* PASCAL operator new(size_t nSize);
void* PASCAL operator new(size_t, void* p);
void PASCAL operator delete(void* p);
#ifdef _DEBUG && !defined(_AFX_NO_DEBUG_CRT)
// for file name/line number tracking using DEBUG_NEW
void* PASCAL operator new(size_t nSize,
                           LPCSTR lpszFileName,
                           Int nLine);
#endif
// Disable the copy constructor and assignment by default
// so you will get compiler errors instead of unexpected
// behavior if you pass objects by value or assign objects.
Protected:
    CObject(const CObject& objectSrc); //no implementation
    void operator=(const CObject& objectSrc);
// Attributes
public
    BOOL IsSerializable() const;
    BOOL IsKindOf(const CRuntimeClass* pClass) const;
// Overridables
    virtual void Serialize(CArchive& ar);
// Diagnostic Support
    virtual void AssertValid() const;

```



```

    virtual void Dump(CDumpContext& dc) const;
// Implementation
public:
    static const AFX_DATA CruntimeClass classCObject;
#ifdef _AFXDLL
    static CRuntimeClass* PASCAL _GetBaseClass();
#endif
};

```

为了清楚起见，我们略微修改了这段代码，但其与在 `afx.h` 头文件中看到的基本上相同。

仔细查看这段代码，注意，**CObject** 类分成公有、保护和私有几部分。**CObject** 类也提供了正常的和动态的类型检验以及串行化。动态类型检验允许在运行期间确定对象类型。对象的状态可以保存在像磁盘这样的存储介质中，在概念上称为暂存。对象暂存也允许对象成员函数暂存，允许恢复对象数据。

子类是由 MFC 父类派生的，例如，**CGdiObject** 类是由 **CObject** 父类派生的。从下面这段程序代码可以查看在 `afxwin.h` 中定义的 **CGdiObject** 类。

```

////////////////////////////////////
// CGdiObject abstract class for CDC SelectObject
class CGdiObject : public CObject
{
    DECLARE_DYNCREATE(CGdiObject)
public:
    // Attributes
    HGDIOBJ m_hObject; // must be first data member
    operator HGDIOBJ() const;
    HGDIOBJ GetSafeHandle() const;
    static CGdiObject* PASCAL FromHandle(HGDIOBJ hObject);
    static void PASCAL DeleteTempMap();
    BOOL Attach(HGDIOBJ hObject);
    HGDIOBJ Detach();
    // Constructors
    CGdiObject(); // must create a derived class object
    BOOL DeleteObject();
    // Operations
    int GetObject(int nCount, LPVOID lpObject) const;
    UINT GetObjectType() const;
    BOOL CreateStockObject(int nIndex);
    BOOL UnrealizeObject();
    BOOL operator==(const CGdiObject& obj) const;
    BOOL operator!=(const CGdiObject& obj) const;
    // Implementation

```

```
public:
    virtual ~CGdiObject();
#ifdef _DEBUG
    virtual void Dump(CDumpContext& dc) const;
    virtual void AssertValid() const;
#endif
};
```

注意, **CGdiObject** 类及其方法(成员函数)允许在 Windows 程序中使用一些绘图项目, 如工具和定制的笔、画刷、创建的字体等。有些类如 **CPen**、**CBrush** 类是从 **CGdiObject** 更进一步派生的。

许多面向过程 Windows 函数调用和其等价类库对象之间的转换是直观的。例如, 在传统的过程 Windows 程序中, **DeleteObject()**函数的调用使用下列语法:

```
DeleteObject(hPen);    /*hPen is the pen handle*/
```

在面向对象程序中, 同样的效果可以使用成员函数获得, 语法如下:

```
newpen.DeleteObject();    //newpen is current pen
```

Microsoft 在开发所有的 Windows 类中使用这种基本方式, 使得从传统过程函数调用到 MFC 类对象调用的转化容易些。在本章的例子程序中, 将看到大量的代码, 由于有过程编码的经验, 我们已经对其熟悉了。

10.3 什么是应用程序向导和类向导

Microsoft 提供了称为 AppWizard(应用程序向导)的动态代码生成器。AppWizard 很依赖 MFC 类库, 用其可产生面向对象的代码。要使用 AppWizard 创建一个工程, 在 Visual C++ 编译器中选择 File|New 菜单选项。然后, 从选项列表中选择 MFC AppWizard。AppWizard 产生一个代码模板, 该模板允许按应用程序的需要选择某些特性。

与 AppWizard 有密切关系的是 ClassWizard(类向导)。ClassWizard 可以用于添加类或定制已存在的类。ClassWizard 仅可在 AppWizard 创建了模板代码之后使用。要启动 ClassWizard, 可以使用 View 菜单中的 ClassWizard 菜单项。

10.4 一个图形程序

在本节将讨论使用 AppWizard 和 ClassWizard 创建基本程序所需要的所有步骤, 使用 AppWizard 需要以一定的次序完成特定的步骤。此处讨论的这些步骤, 将从头到尾完成第一个名为 graph 的程序。



启动 Visual C++ 编译器，和我们一起完成创建 Graph 程序的步骤。

10.4.1 使用 AppWizard

使用 Visual C++ 菜单栏，选择 File|New 菜单项，出现一个如图 10-1 所示的对话框。通过选择 MFC AppWizard(exe)选项，允许我们开始一个新的工程。

新的工程命名为 graph，如图 10-1 所示。为工程命名之后，就开始了使用 AppWizard 开发程序阶段。查看图 10-1，注意选择这一工程的存放位置。接下来，单击 OK 按钮，启动 AppWizard。

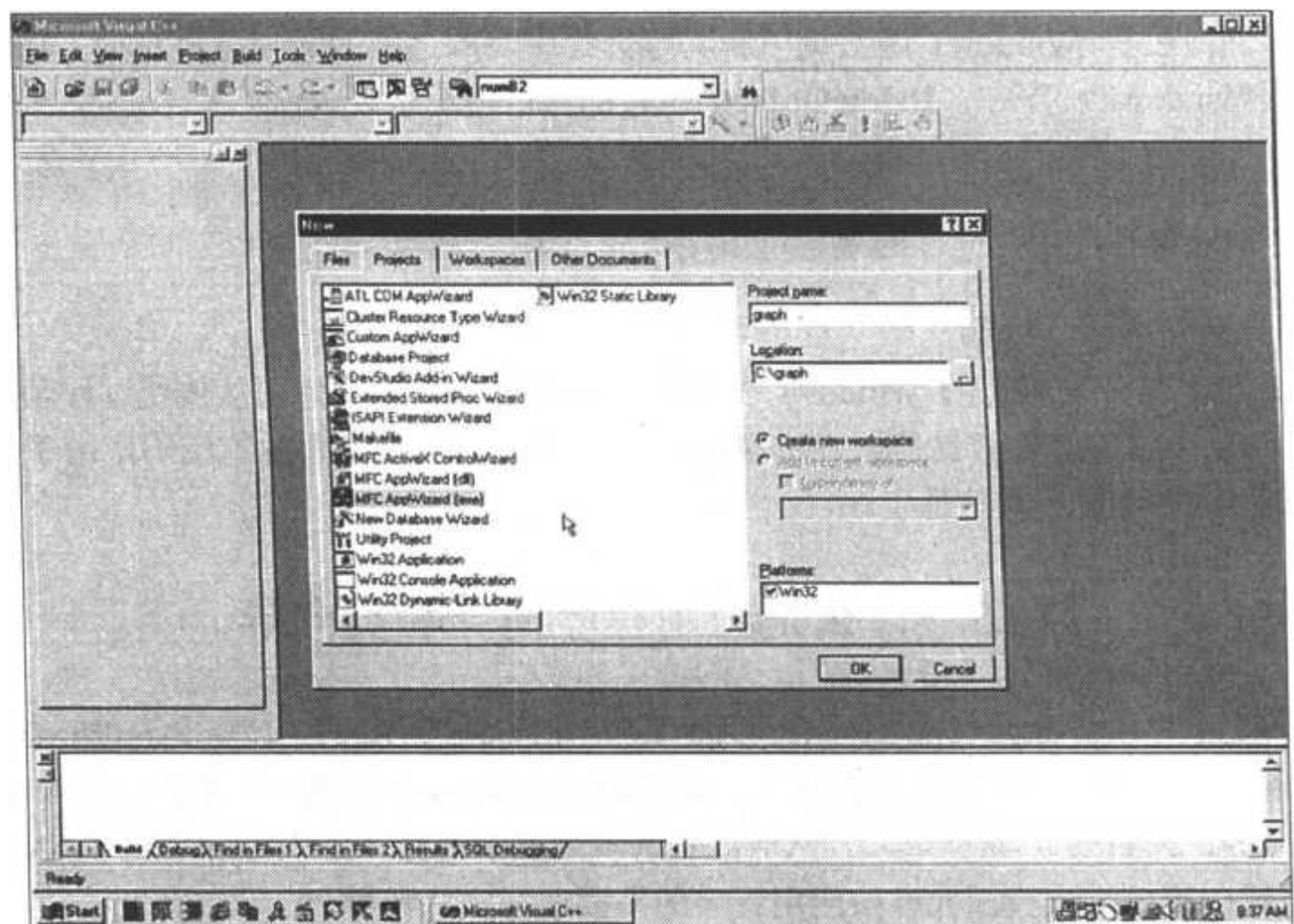


图 10-1 File|New 菜单项为一个新工程提供了进入 MFC AppWizard 的方法

AppWizard 产生工程的第一步涉及到确定工程是处理单文档、多文档，还是对话框，如图 10-2 所示。

单文档界面是最简单的，本例不需要多文档。确定 Document/View architecture support 是选中的。接受缺省的资源语言选择。单击“Next”按钮开始第二步，如图 10-3 所示。

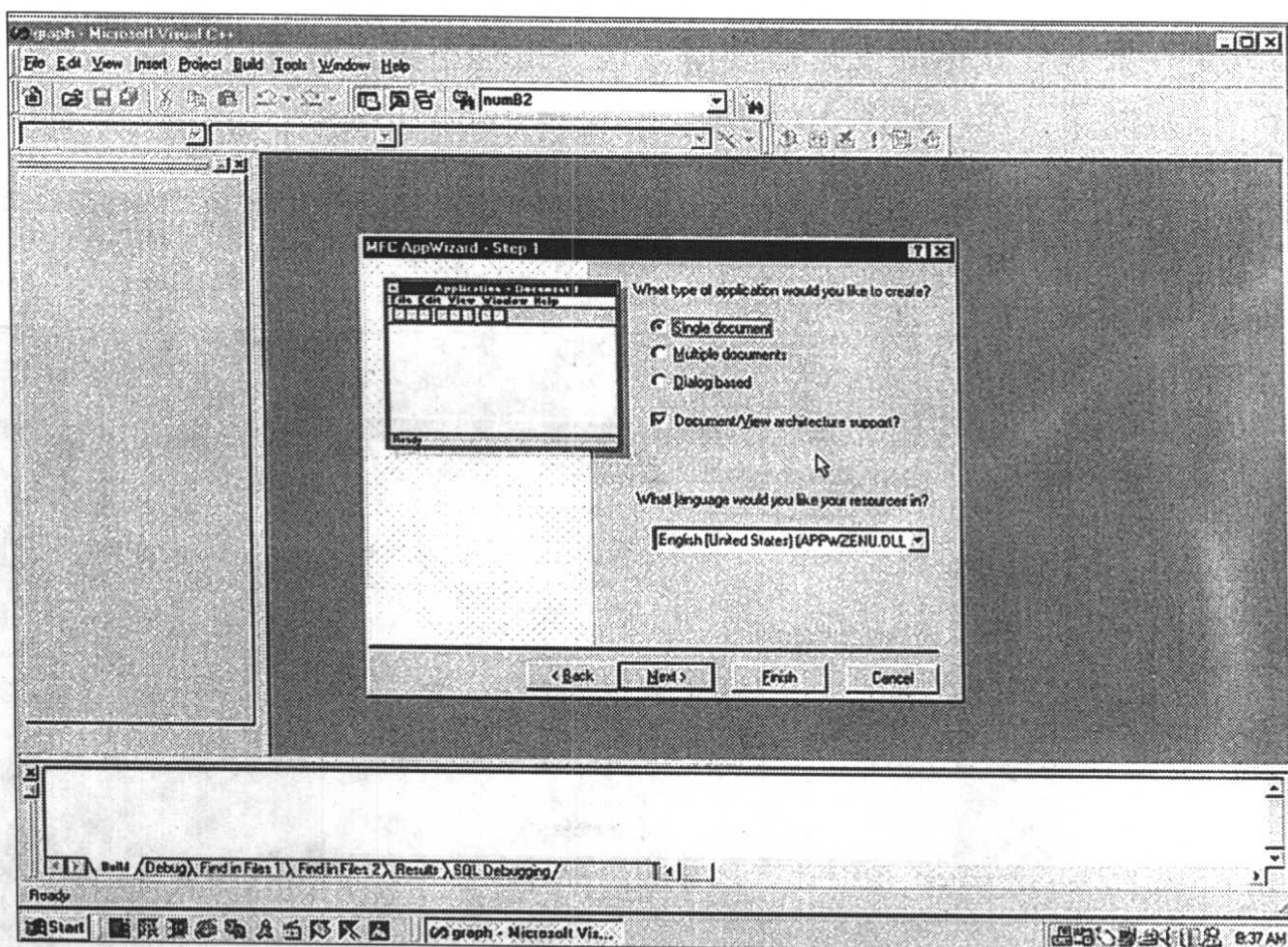


图 10-2 Step 1—这一工程使用单文档界面

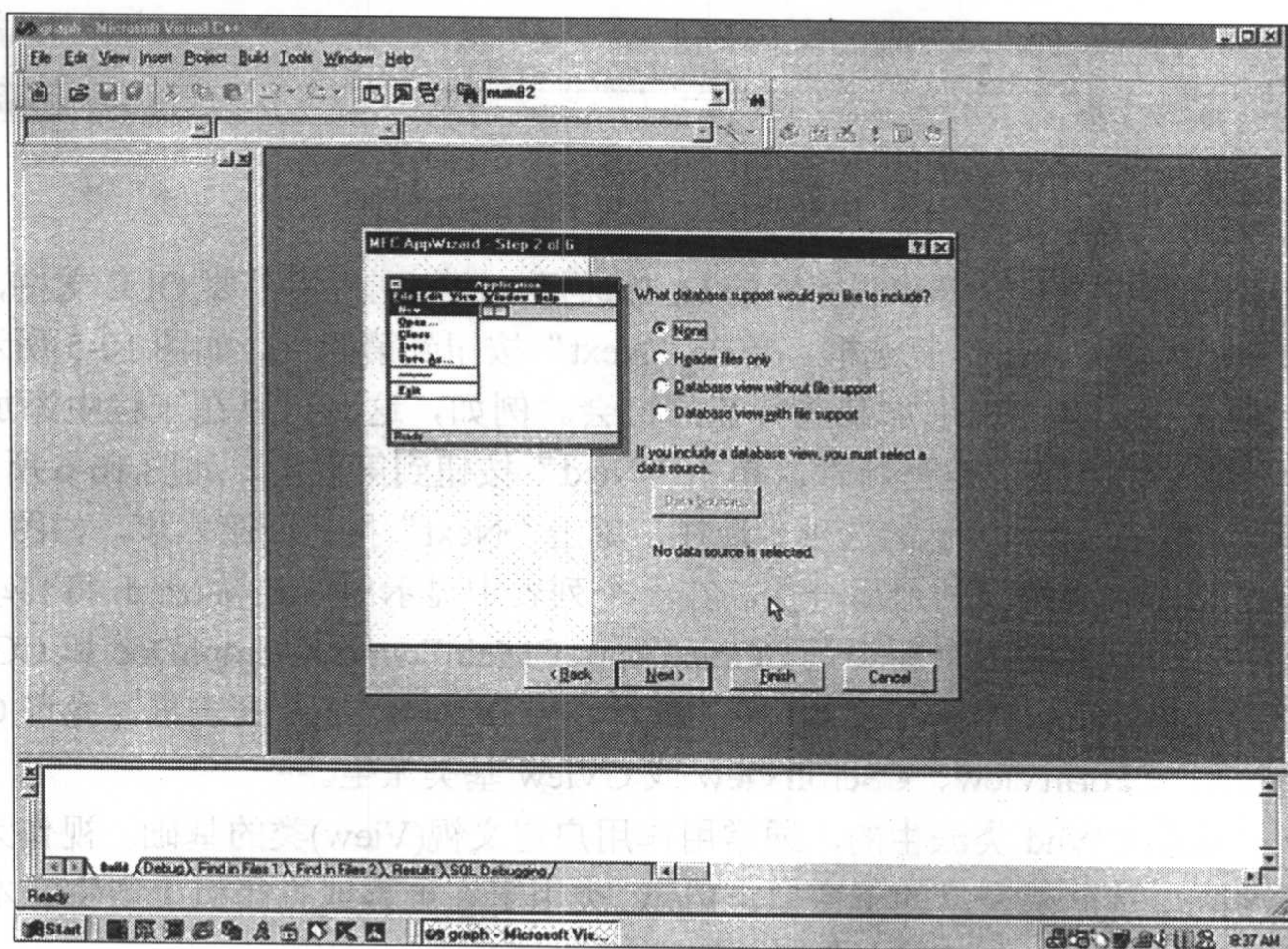


图 10-3 Step 2—因为不需要数据库支持，所以选择 None



第二步只是在要包含数据库支持时使用。这一工程不需要数据库支持，所以选择 None，单击“Next”按钮到第三步，如图 10-4 所示。

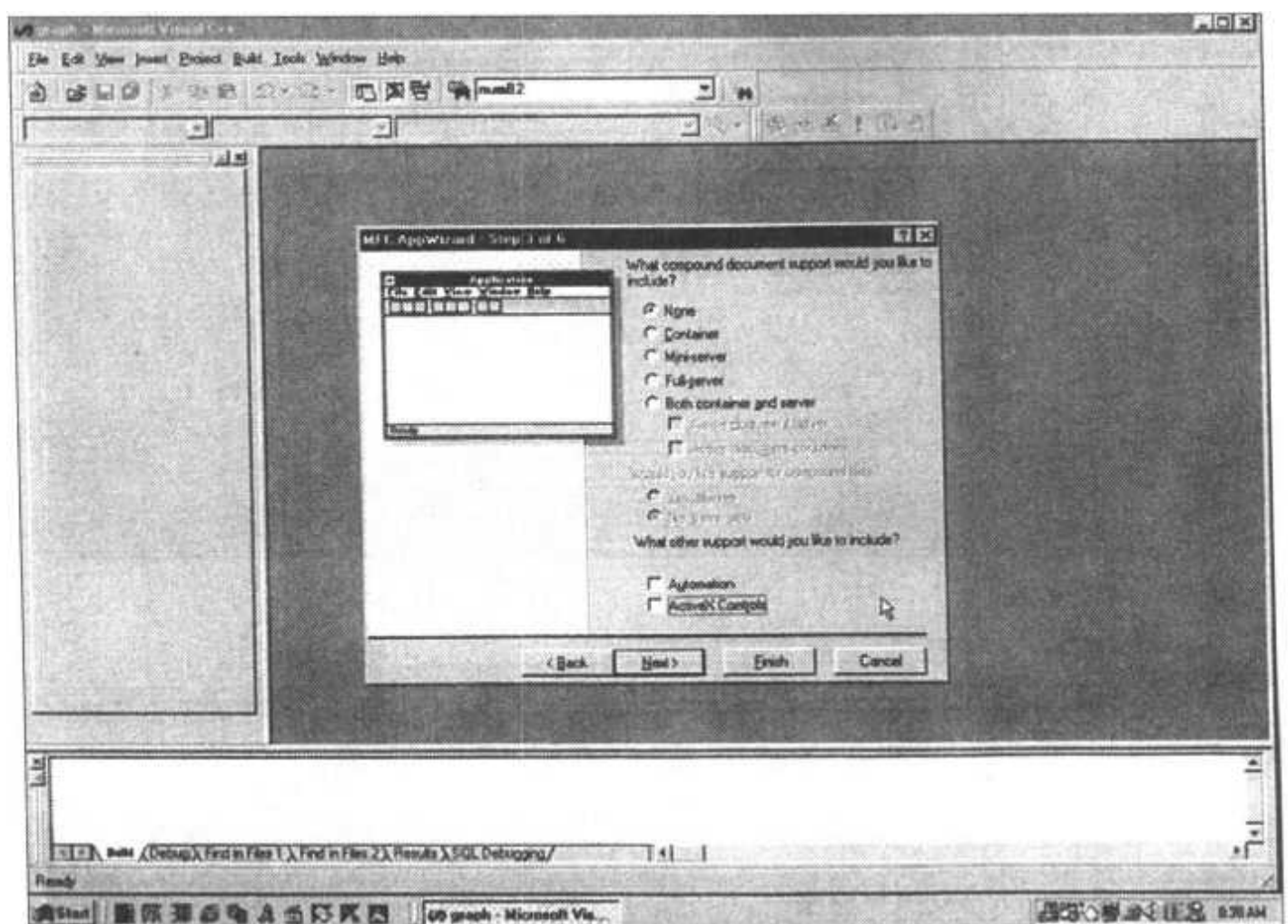


图 10-4 Step 3—这一工程不需要复合文档支持

第三步允许选择具有容器和服务器的 OLE 选择。这一工程不需要 OLE 支持，所以选择 None。关闭 ActiveX Controls 复选框，单击“Next”按钮到第四步，如图 10-5 所示。

第四步提供了一了在工程中加某些特征的机会。例如，这时可以在工程中添加工具栏和状态栏等。这一工程不需要这些特性。单击“Next”按钮到第五步，如图 10-6 所示。

第五步选择如图 10-6 所示的选项。现在，单击“Next”按钮到第六步，如图 10-7 所示。

第六步是指定工程特性的最后一步。在一个列表中显示出 AppWizard 将自动产生的新类。对于这一程序，产生的四个新类是 CGraphApp、CMainFrame、CGraphDoc 和 CGraphView。

从列表中选择 CGraphView，如图 10-7 所示。展开 Base Class 列表框，允许 CGraphView 类从 CEditView、CFormView、CScrollView 或 CView 基类派生。

CView 类是从 CWnd 类派生的，通常用作用户定义视(View)类的基础。视作为文档和用户之间的缓冲区，是框架窗口的子窗口。View 类用于在屏幕或打印机上产生文本的映象。这些类用键盘和鼠标输入作为对文档的操作。

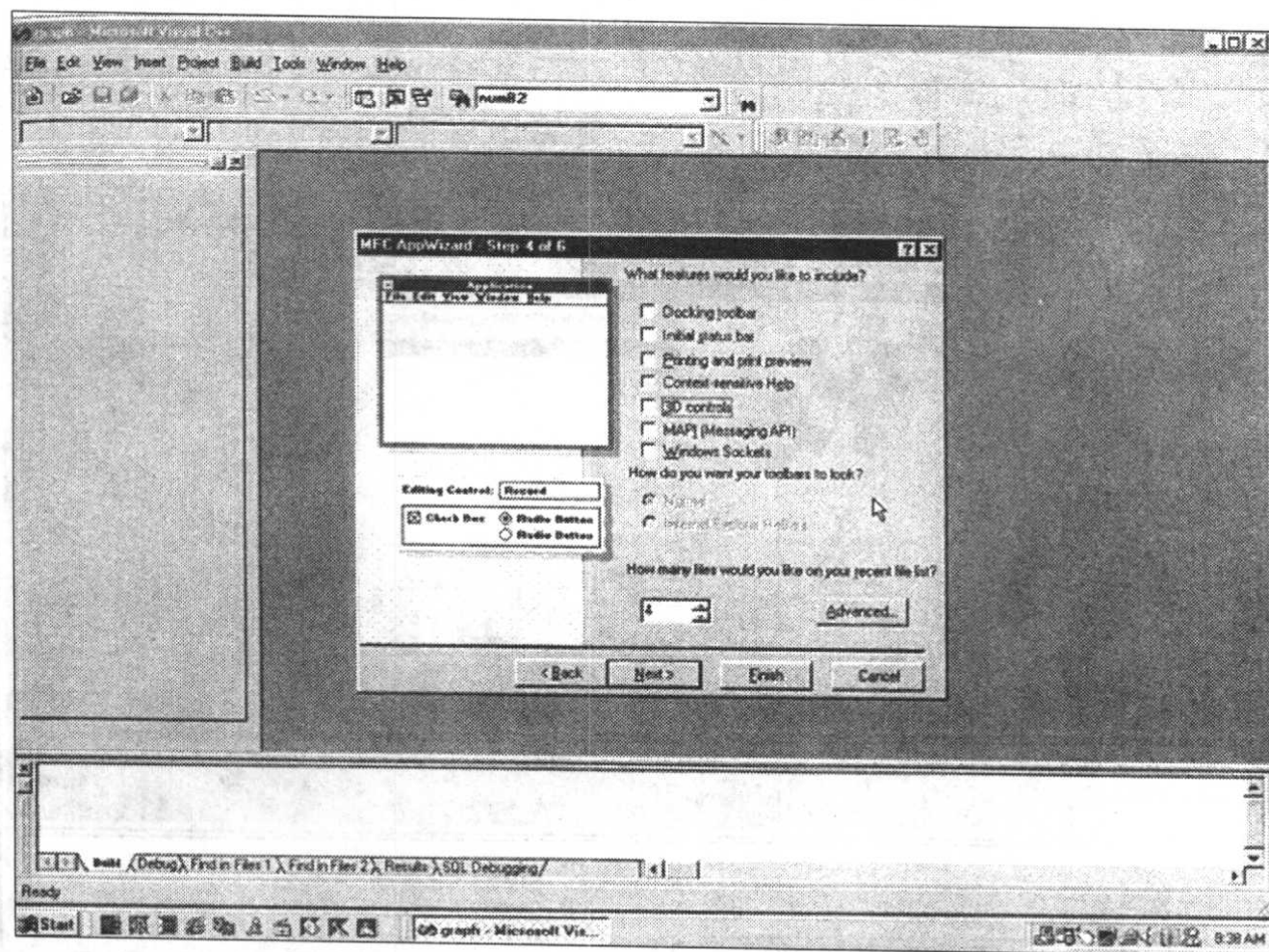


图 10-5 Step 4—允许在工程中附加一些特殊风格

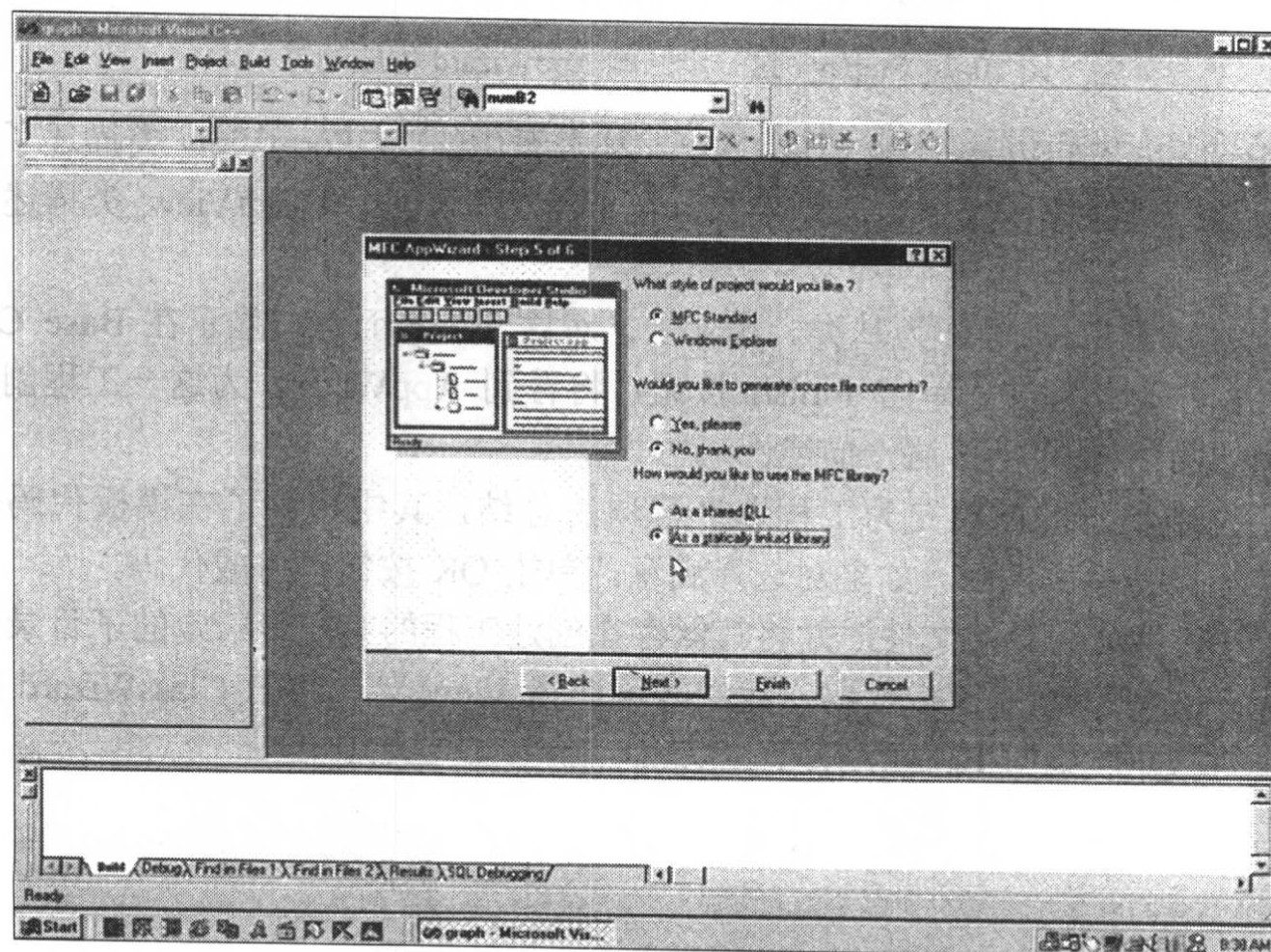


图 10-6 Step 5—确定 MFC 工程类型，可以包含源文件注释和指定 MFC 类库

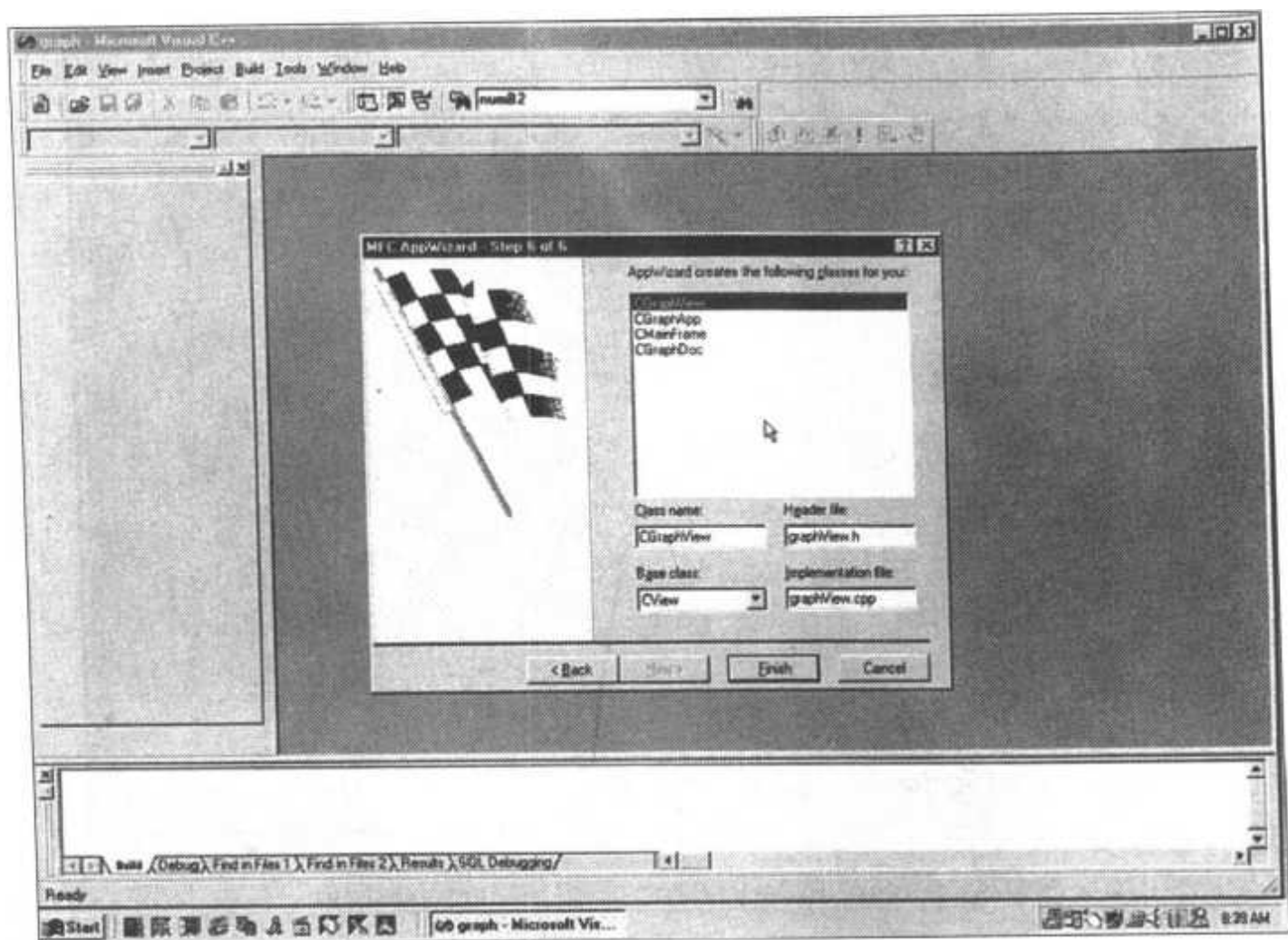


图 10-7 Step 6—这一步回顾 AppWizard 生成的类

CFormView 和 CEditView 类也是从 CView 基类派生的。CFormView 类描述了一个可滚动的视，该视是以对话框模板资源为基础的，包括对话框控件。CEditView 类描述了一个文本编辑器。

这一工程将以 CView 类作为基类。实际上，对这一工程而言，显示在 Base Class 列表框中的类都可以作为其基类。单击 Finish 按钮，将看到 AppWizard 为这一工程创建的项目的摘要说明。如图 10-8 显示了这一描述性汇总。

这一对话框中显示的信息是为工程选择选项的总结，其中提供了在模板代码生成之前改变前面所做选择的最后机会。如果对选择满意，单击 OK 按钮以生成代码。

AppWizard 将生成大量的文件，并将其保存在创建工程的第一步给定的子目录中。

一旦 AppWizard 生成了模板代码，我们即可选择 View 菜单中的 ClassWizard 选项，为工程添加其他的功能，如图 10-9 所示。

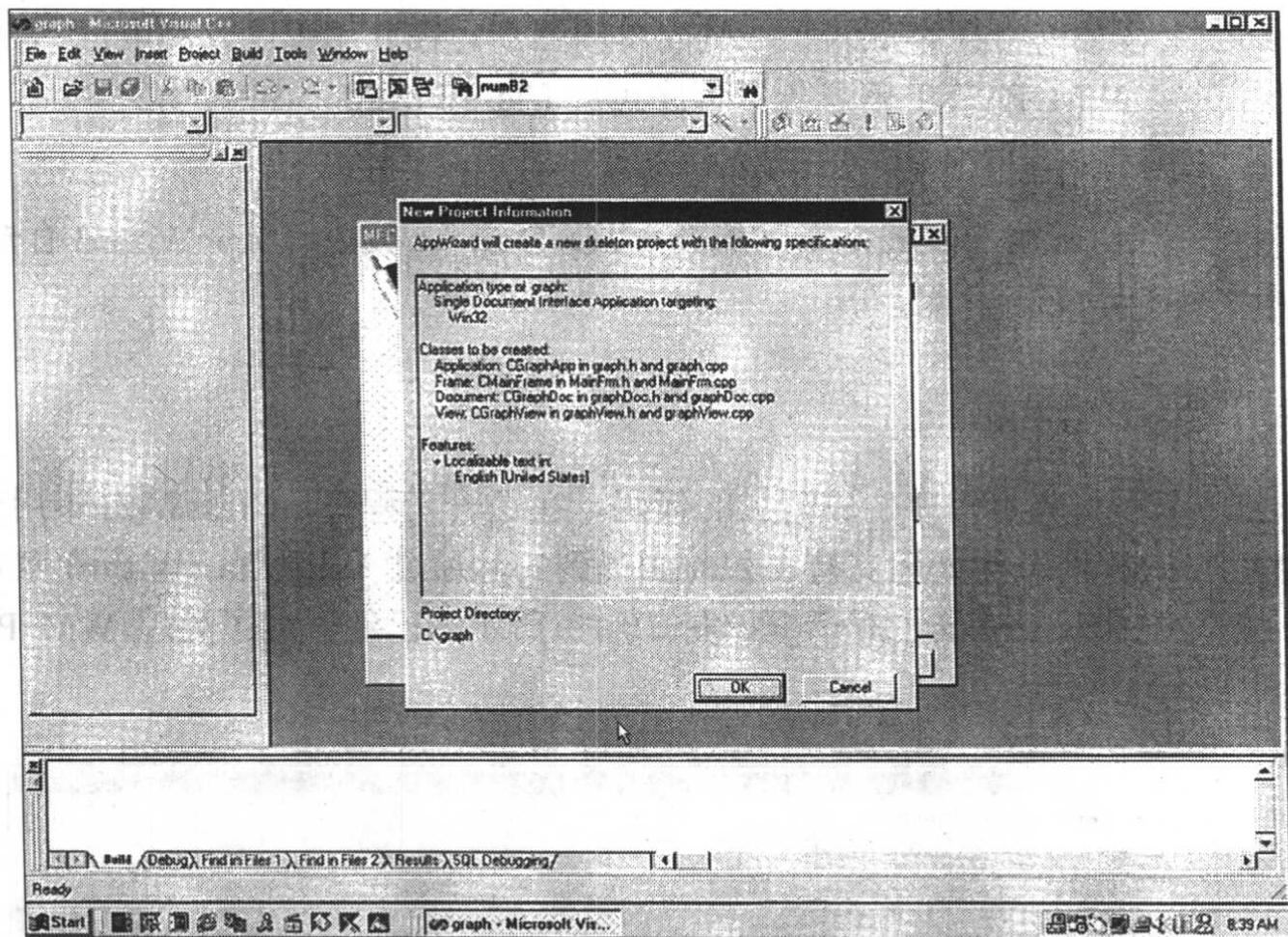


图 10-8 AppWizard 提供一个关于为此工程生成的项目的总结

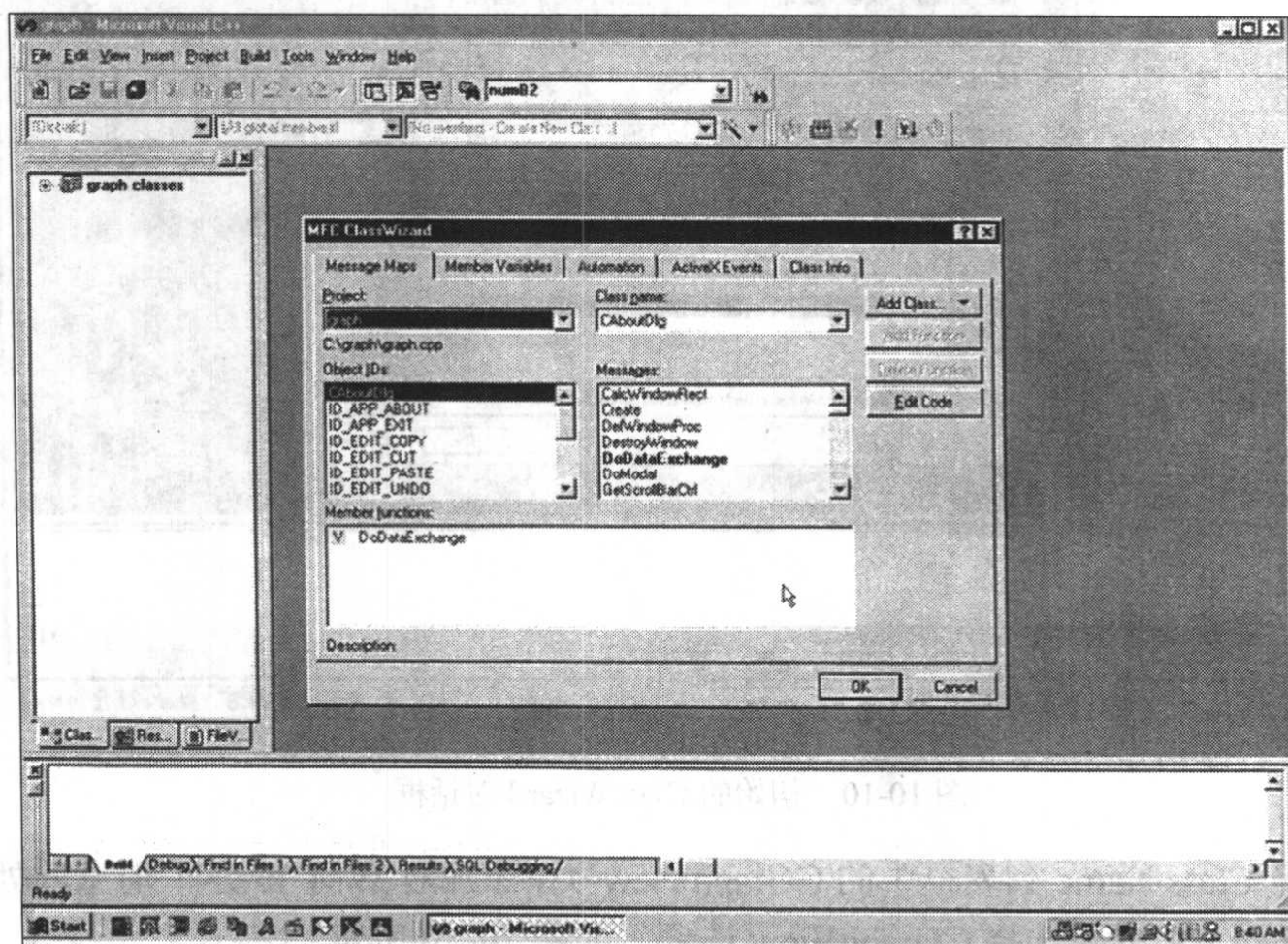


图 10-9 由 AppWizard 生成的模板代码可以使用 ClassWizard 自定义



这一 graph 工程最终将在客户区画一些简单的图形。在此工程中，将响应 WM_PAINT 消息画出各种图形。WM_PAINT 消息处理程序可用 ClassWizard 添加到程序中。

设计提示

许多图形程序可以加到 OnDraw 成员函数中，OnDraw 函数是由 AppWizard 自动添加的，有了它后不再需要 OnPaint() 函数。

10.4.2 使用 ClassWizard

ClassWizard 用于为应用程序生成附加的代码。在这一工程中，ClassWizard 用于在程序中添加对 WM_PAINT 消息的处理支持。如前面提到的，通过 View|ClassWizard 菜单项启动 ClassWizard。初始的 ClassWizard 对话框如图 10-10 所示，已经添加了处理 WM_PAINT 消息的 OnPaint() 成员函数。

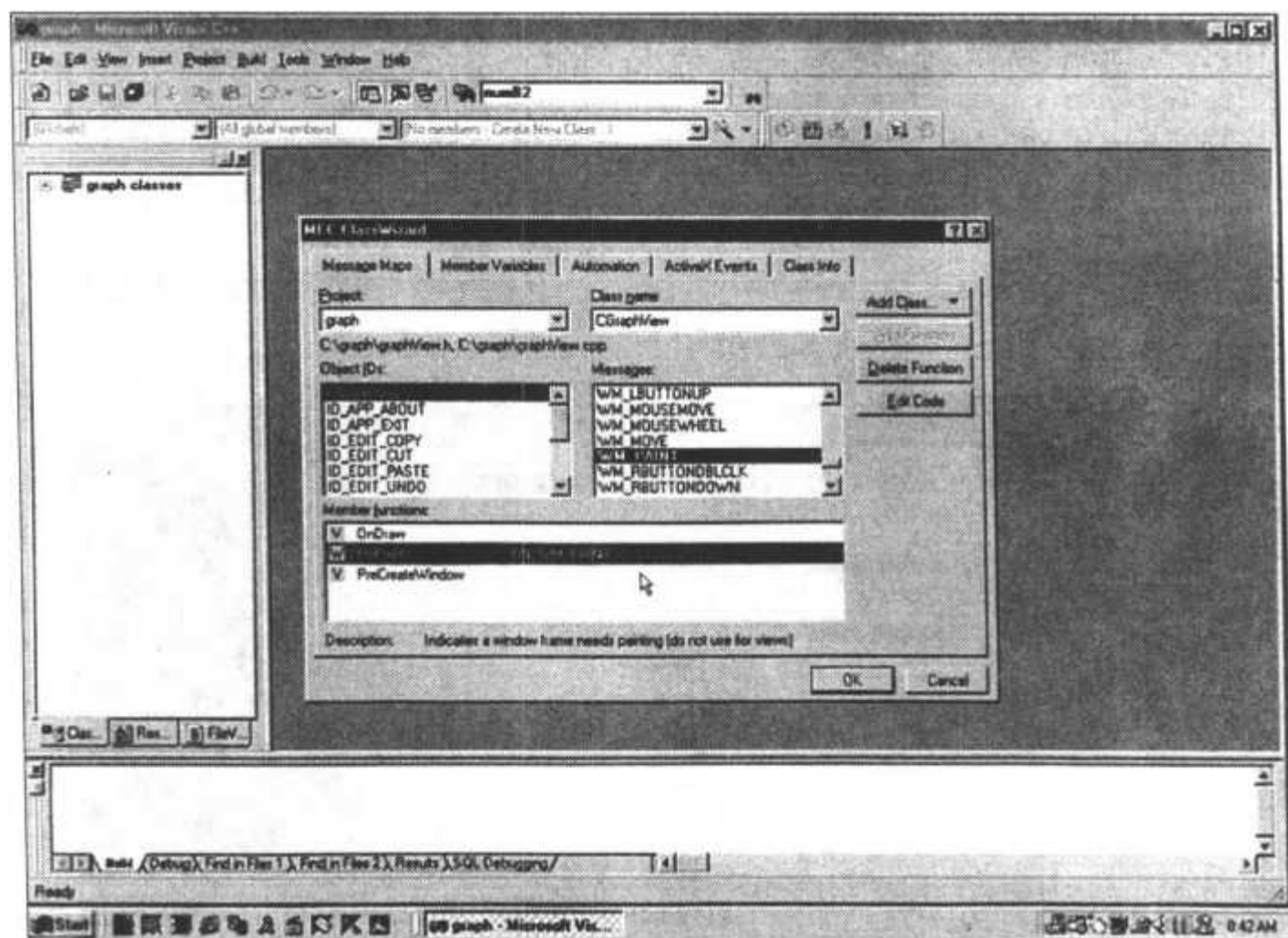


图 10-10 初始的 ClassWizard 对话框

通过选择 Class Name 列表框中的 CGraphView 类，添加对 WM_PAINT 消息的处理支持。然后，从 Object IDs 列表选择 CGraphView 选项，如图 10-10 所示。

从 Class Name 列表框中选择了 CGraphView 后，大量的 Windows 消息将在 Messages 列

表框中显示。从 Messages 列表中选择 WM_PAINT 消息, OnPaint()成员函数将出现在 Member Function 列表中。

现在, 在 graphview.cpp 中包含插入的 OnPaint()代码, 如图 10-11 所示。

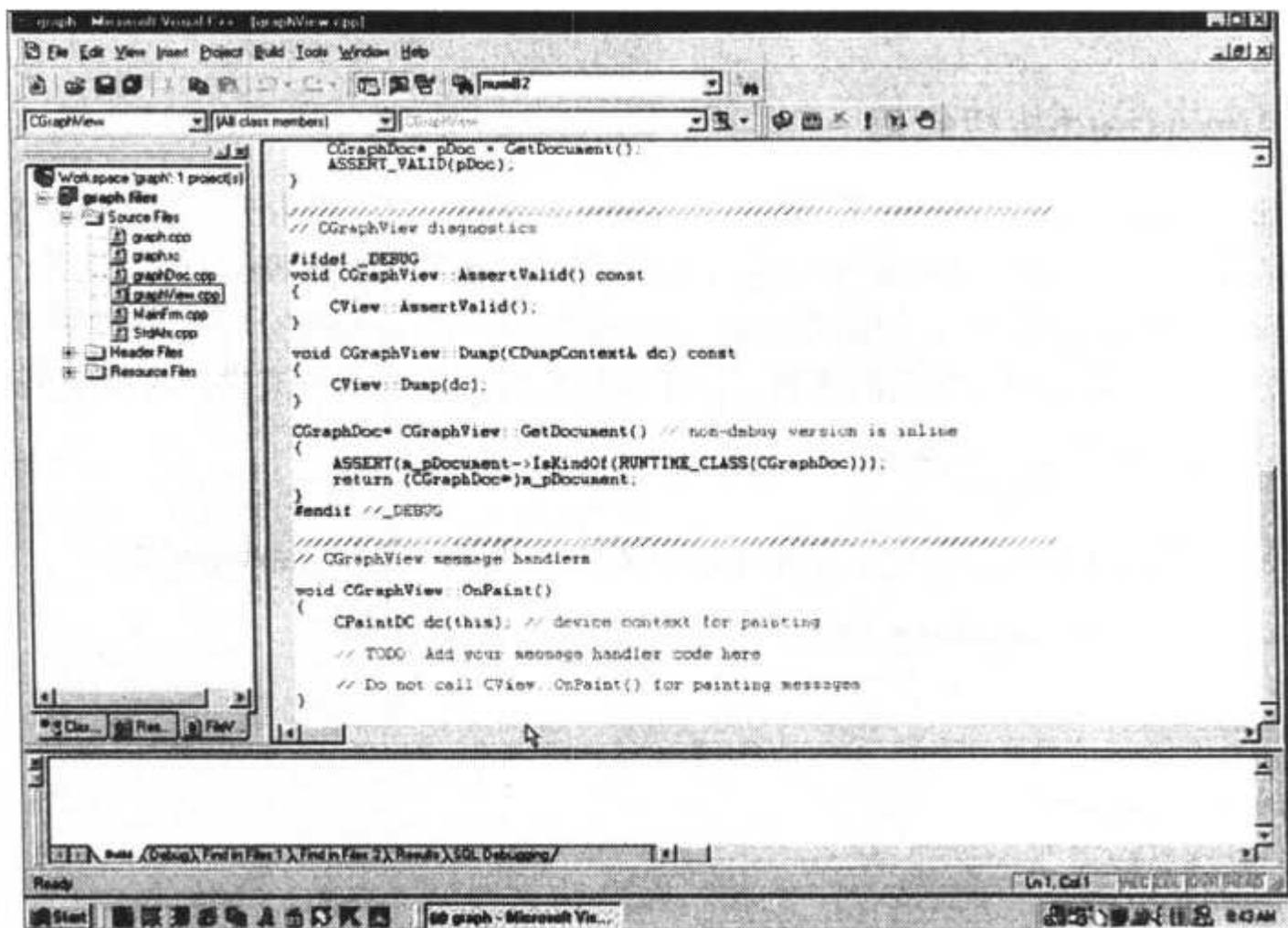


图 10-11 添加到工程中的基本 OnPaint()成员函数

为了完善这一程序, 可以添加各种图形处理到 OnPaint()成员函数中。本章稍后, 将介绍如何操作。下一步是编译和测试 AppWizard 模板代码。

10.4.3 建立 AppWizard 代码

当已经添加所需要的所有消息处理程序, 如 WM_PAINT 消息处理后, ClassWizard 产生的程序代码可以创建执行程序。使用 Build|Rebuild ALL 菜单项, 可以建立执行程序,

在创建执行程序过程中, 将编译和连接 graph.cpp、mainfrm.cpp、graphicdoc.cpp 和 graphicview.cpp 四个源代码文件。当编译完成后, 查看保存这些文件的子目录, 将看到有 30 多个文件保存在此处, 其中包括与其有关的头文件。

可执行文件在工程的 Debug 子目录中。因为此时还没有添加对图形的处理, 所以开始执行程序, 屏幕是空白的。程序的窗口有一个基本的菜单, 但绝大多数菜单项不起作用, 这



是因为这些菜单项的处理代码必须由程序员自己添加。

设计提示

要清除工程中由 AppWizard 添加的我们不需要的菜单项，应使用 Resource Editor。这样使程序更清晰。

10.4.4 AppWizard 模板代码

AppWizard 加上 ClassWizard 的帮助，产生了初始图形程序的四个重要的 C++ 文件。这些文件名称为：graph.cpp、MainFrm.cpp、graphDoc.cpp 和 graphView.cpp。这几个文件都有其相应的头文件：graph.h、MainFrm.h、graphDoc.h 和 graphView.h。在这些头文件中包含了在每个 *.cpp 文件中特定类的声明。为了简短地讨论每个 C++ 文件，考察下列程序。

10.4.4.1 graph.cpp 文件

下面显示的 graph.cpp 文件是作为程序的主文件，其中包含 CGraphApp 类。

```
// graph.cpp : Defines class behaviors for the application.
//
#include "stdafx.h"
#include "graph.h"
#include "MainFrm.h"
#include "graphDoc.h"
#include "graphView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CGraphApp
BEGIN_MESSAGE_MAP(CGraphApp, CWinApp)
//{{AFX_MSG_MAP(CGraphApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    // NOTE - the ClassWizard will add and remove mapping
    // Macros here.
    // DO NOT EDIT what you see in these blocks of
    // generated code
//}}AFX_MSG_MAP
// Standard file based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```



```

END_MESSAGE_MAP()
// CGraphApp construction
CGraphApp::CGraphApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}
// The one and only CGraphApp object
CGraphApp theApp;
// CGraphApp initialization
BOOL CGraphApp::InitInstance()
{
    // Standard initialization
    // If you are not using these features and wish to reduce
    // the size of your final executable, you should remove
    // from the following the specific initialization routines
    // you do not need.
    // Change the registry key under which our settings are
    // stored. You should modify this string to be something
    // appropriate such as the name of your company or
    // organization.
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));
    LoadStdProfileSettings(); // Load INI file options
                               (including MRU)
    // Register the application's document templates. Document
    // templates serve as the connection between documents,
    // frame windows and views.
    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CGraphDoc),
        RUNTIME_CLASS(CMainFrame), // main SDI frame window
        RUNTIME_CLASS(CGraphView));
    AddDocTemplate(pDocTemplate);
    // Parse command line for standard shell commands, DDE, file open
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);
    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;
}

```



```
// The one and only window is initialized - show and update
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
return TRUE;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About
class CAboutDlg : public CDialog
{
public:
    CAboutDlg();
// Dialog Data
    //{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}AFX_DATA
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CAboutDlg)
    protected:
        virtual void DoDataExchange(CDataExchange* pDX);
    //}AFX_VIRTUAL
// Implementation
protected:
    //{AFX_MSG(CAboutDlg)
    // No message handlers
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{AFX_DATA_INIT(CAboutDlg)
    //}AFX_DATA_INIT
}
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{AFX_DATA_MAP(CAboutDlg)
    //}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{AFX_MSG_MAP(CAboutDlg)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()
// App command to run the dialog
```

```
void CGraphApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}
////////////////////////////////////
// CGraphApp commands
```

程序清单开头的消息映射属于 CGraphApp 类，这一消息映射明确地将 ID_APP_ABOUT、ID_FILE_NEW 和 ID_FILE_OPEN 消息连接到其成员函数 OnAppAbout()、CWinApp::OnFileNew()和 CWinApp::OnFileOpen()上。要注意程序清单中的构造函数、初始化函数 InitInstance()和成员函数 OnAppAbout 的实现。

About 对话框是从 CDialog 类派生的。查看代码的后部分，将注意到消息映射、构造函数和此派生类成员函数 CDialog::DoDataExchange()。

如同从程序结尾看到的，没有初始的 CGraphApp 命令。

10.4.4.2 mainfrm.cpp 文件

下面显示的是 mainfrm.cpp 文件，其包含了 CMainFrame 框架类。此类是从 CFrameWnd 类派生的，用于控制全部的单文档界面(SDI)框架特性。

```
// MainFrm.cpp : implementation of the CMainFrame class
//
#include "stdafx.h"
#include "graph.h"
#include "MainFrm.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////
// CMainFrame
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    // NONT - the ClassWizard will add and remove mapping
    // macros here.
    // DO NOT EDIT what you see in these blocks of
    // generated code !
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```



```
////////////////////////////////////
// CMainFrame construction/destruction
CMainFrame::CMainFrame()
{
    // TODO : add member initialization code here
}
CMainFrame::~CMainFrame()
{
}
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    //TODO : Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs
    return TRUE;
}
////////////////////////////////////
// CMainFrame diagnostics
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}
#endif // _DEBUG
////////////////////////////////////
// CMainFrame message handlers
```

起初的消息映射、构造函数和析构函数不含代码，成员函数 `AssertValid()` 和 `Dump()` 使用的是父类中的定义。注意 `CMainFrame` 初始情况下是没有消息处理程序的。

10.4.4.3 graphdoc.cpp 文件

此处显示的是 `graphdoc.cpp` 文件，其包含了 `CGraphDoc` 类，对于此程序来说是唯一的。这一文件用于保留加载和保存的文档数据。

```
// graphDoc.cpp : implementation of the CGraphDoc class
//
#include "stdafx.h"
```

```

#include "graph.h"
#include "graphDoc.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CGraphDoc
IMPLEMENT_DYNCREATE(CGraphDoc, CDocument)
BEGIN_MESSAGE_MAP(CGraphDoc, CDocument)
    //{AFX_MSG_MAP(CGraphDoc)
    // NOTE - add and remove mapping macros here.
    // DO NOT EDIT these blocks of generated code!
    //}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CGraphDoc construction/destruction
CGraphDoc::CGraphDoc()
{
    // TODO: add one-time construction code here
}
CGraphDoc::~CGraphDoc()
{
}
BOOL CGraphDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
    return TRUE;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CGraphDoc serialization
void CGraphDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {

```



```
        // TODO: add loading code here
    }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CGraphDoc diagnostics
#ifdef _DEBUG
void CGraphDoc::AssertValid() const
{
    CDocument::AssertValid();
}
void CGraphDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif // _DEBUG
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CGraphDoc commands
```

检查这段程序，我们注意到，在消息映射、构造函数和析构函数中没有代码。有四个成员函数可以提供重要的文档支持。OnNewDocument()成员函数使用的是父类的定义。Serialize()成员函数支持驻留的对象。在我们第二个程序例子中将使用这一成员函数帮助文件输入/输出。成员函数 AssertValid()和 Dump()使用父类中的定义，没有初始 CGraphDoc 命令。

10.4.4.4 Graphview.cpp 文件

此处显示的 graphview.cpp 文件提供了文档视，在这一实现中，CGraphView 类是从 CView 类派生的。CGraphView 对象常用于查看 CGraphDoc 对象。

```
// GraphView.cpp : implementation of the CGraphView class
//
#include "stdafx.h"
#include "graph.h"
#include "graphDoc.h"
#include "graphView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CGraphView
IMPLEMENT_DYNCREATE(CGraphView, CView)
```

```

BEGIN_MESSAGE_MAP(CGraphView, CView)
   //{{AFX_MSG_MAP(CGraphView)
    ON_WM_PAINT()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////
// CGraphView construction/destruction
CGraphView::CGraphView()
{
    // TODO: add construction code here
}
CGraphView::~CGraphView()
{
}
BOOL CGraphView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class / styles here by modifying
    // the CREATESTRUCT cs
    return CView::PreCreateWindow(cs);
}
/////////////////////////////////////////////////////////////////
// CGraphView drawing
void CGraphView::OnDraw(CDC* pDC)
{
    CGraphDoc* pDoc = GetDocument();
    // TODO: add draw code for native data here
    ASSERT_VALID(pDoc);
}
/////////////////////////////////////////////////////////////////
// CGraphView diagnostics
#ifdef _DEBUG
void CGraphView::AssertValid() const
{
    CView::AssertValid();
}
void CGraphView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}
CGraphDoc* CGraphView::GetDocument() // non-debug version
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CGraphDoc)));
    return (CGraphDoc*)m_pDocument;
}

```



```
}
#endif // _DEBUG
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CGraphView message handlers
void CGraphView::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // TODO: Add your message handler code here
    // Do not call CView::OnPaint() for painting messages
}
```

如果我们刚使用 AppWizard 生成了这一程序，消息映射应该是空的。然而，要记住 ClassWizard 用于添加 WM_PAINT 消息处理程序的能力。CGraphView 对象的构造函数和析构函数是空的。

OnDraw()成员函数使用指针 pDoc 指向文本，AssertValid()和 Dump()成员函数使用父类的定义。在这一程序清单的最后描述的是消息处理程序 OnPaint()，在此处可以插入一些简单的图形处理函数像 LineTo()、Rectangle()和 Ellipse()等。

错误监视

在 OnDraw()和 OnPaint()成员函数之间来回移动 GDI 函数时要注意，由于实现的方式问题，这些函数的调用语法略有不同。例如，在 OnPaint()函数中，Rectangle()函数的调用为：

```
dc.Rectangle(10,20,40,80);
```

在 OnDraw()成员函数中，同样的函数调用使用下列格式：

```
pdc->Rectangle(10,20,40,80);
```

10.4.5 在客户区的图形对象

在这个工程开发过程中，使用 AppWizard 和 ClassWizard 建立单文档(SDI)界面应用程序，视类是由父类 CView 派生的。此处 ClassWizard 将 WM_PAINT 消息处理程序添加到了 AppWizard 模板代码中。

OnPaint 方法是在客户区绘制简单图形对象的理想场所。我们将看到需要添加一些代码。下面的程序清单显示了 graph 工程中添加到 OnPaint()方法中的代码：

```
void CGraphView::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // TODO: Add your message handler code here
```



```
// Do not call CView::OnPaint() for painting messages
static DWORD dwColor[9]={RGB(0,0,0),      //black
                        RGB(255,0,0),     //red
                        RGB(0,255,0),     //green
                        RGB(0,0,255),     //blue
                        RGB(255,255,0),   //yellow
                        RGB(255,0,255),   //magenta
                        RGB(0,255,255),   //cyan
                        RGB(127,127,127), //gray
                        RGB(255,255,255)}; //white

POINT poly1pts[4],polygpts[5];
int xcoord;
CBrush newbrush;
CBrush* oldbrush;
CPen newpen;
CPen* oldpen;

// draws and fills a circle
newpen.CreatePen(PS_SOLID,2,dwColor[3]);
oldpen=dc.SelectObject(&newpen);
newbrush.CreateSolidBrush(dwColor[3]);
oldbrush=dc.SelectObject(&newbrush);
dc.Ellipse(400,20,650,270);
dc.TextOut(500,150,"circle",6);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
dc.SelectObject(oldpen);
newpen.DeleteObject();
// draws and fills an ellipse
newpen.CreatePen(PS_SOLID,2,dwColor[1]);
oldpen=dc.SelectObject(&newpen);
newbrush.CreateSolidBrush(dwColor[1]);
oldbrush=dc.SelectObject(&newbrush);
dc.Ellipse(325,300,425,250);
dc.TextOut(260,265,"ellipse",7);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
dc.SelectObject(oldpen);
newpen.DeleteObject();
// draws several pixels
for(xcoord=500;xcoord<600;xcoord+=5)
    dc.SetPixel(xcoord,400,0L);
dc.TextOut(540,410,"pixels",6);
// draws a wide diagonal line
newpen.CreatePen(PS_SOLID,10,dwColor[0]);
oldpen=dc.SelectObject(&newpen);
dc.MoveTo(20,20);
```



```
dc.LineTo(100,100);
dc.TextOut(60,20,"<- diagonal line",16);
dc.SelectObject(oldpen);
newpen.DeleteObject();
// draws an arc
newpen.CreatePen(PS_DASH,1,dwColor[3]);
oldpen=dc.SelectObject(&newpen);
dc.Arc(25,125,175,225,175,225,100,125);
dc.TextOut(50,150,"small arc ->",12);
dc.SelectObject(oldpen);
newpen.DeleteObject();
// draws a wide chord
newpen.CreatePen(PS_SOLID,10,dwColor[2]);
oldpen=dc.SelectObject(&newpen);
dc.Chord(125,125,275,225,275,225,200,125);
dc.TextOut(280,150,"<- chord",8);
dc.SelectObject(oldpen);
newpen.DeleteObject();
// draws a pie slice and fills
newpen.CreatePen(PS_SOLID,2,dwColor[0]);
oldpen=dc.SelectObject(&newpen);
newbrush.CreateSolidBrush(dwColor[2]);
oldbrush=dc.SelectObject(&newbrush);
dc.Pie(200,0,300,100,200,50,250,100);
dc.TextOut(260,80,"<- pie wedge",12);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
dc.SelectObject(oldpen);
newpen.DeleteObject();
// draws a rectangle and fills
newbrush.CreateSolidBrush(dwColor[7]);
oldbrush=dc.SelectObject(&newbrush);
dc.Rectangle(25,350,150,425);
dc.TextOut(50,440,"rectangle",9);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();

// draws a rounded rectangle and fills
newbrush.CreateHatchBrush(HS_CROSS,dwColor[3]);
oldbrush=dc.SelectObject(&newbrush);
dc.RoundRect(400,320,550,360,20,20);
dc.TextOut(410,300,"rounded rectangle",17);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// draws a wide polygon and fills
newpen.CreatePen(PS_SOLID,5,dwColor[6]);
oldpen=dc.SelectObject(&newpen);
```

```
newbrush.CreateHatchBrush(HS_FDIAGONAL,dwColor[4]);
oldbrush=dc.SelectObject(&newbrush);
polygpts[0].x=40;
polygpts[0].y=200;
polygpts[1].x=100;
polygpts[1].y=270;
polygpts[2].x=80;
polygpts[2].y=290;
polygpts[3].x=20;
polygpts[3].y=220;
polygpts[4].x=40;
polygpts[4].y=200;
dc.Polygon(polygpts,5);
dc.TextOut(80,230,"<- polygon",10);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
dc.SelectObject(oldpen);
newpen.DeleteObject();
// draws several wide lines with polyline
newpen.CreatePen(PS_SOLID,4,dwColor[5]);
oldpen=dc.SelectObject(&newpen);
polylpts[0].x=210;
polylpts[0].y=330;
polylpts[1].x=210;
polylpts[1].y=400;
polylpts[2].x=250;
polylpts[2].y=400;
polylpts[3].x=210;
polylpts[3].y=330;
dc.Polyline(polylpts,4);
dc.TextOut(250,350,"polyline",8);
dc.SelectObject(oldpen);
newpen.DeleteObject();
}
```

一个有经验的程序员，对绝大多数 GDI 图形绘制是熟悉的。再次编译和运行添加了这些代码的工程。屏幕看上去将如图 10-12 所示。

AppWizard 创建的模板包含 File、Edit 和 Help 菜单，以及各种形状的图形。任何没有实现的菜单项都从此工程的最终版本中清除了。

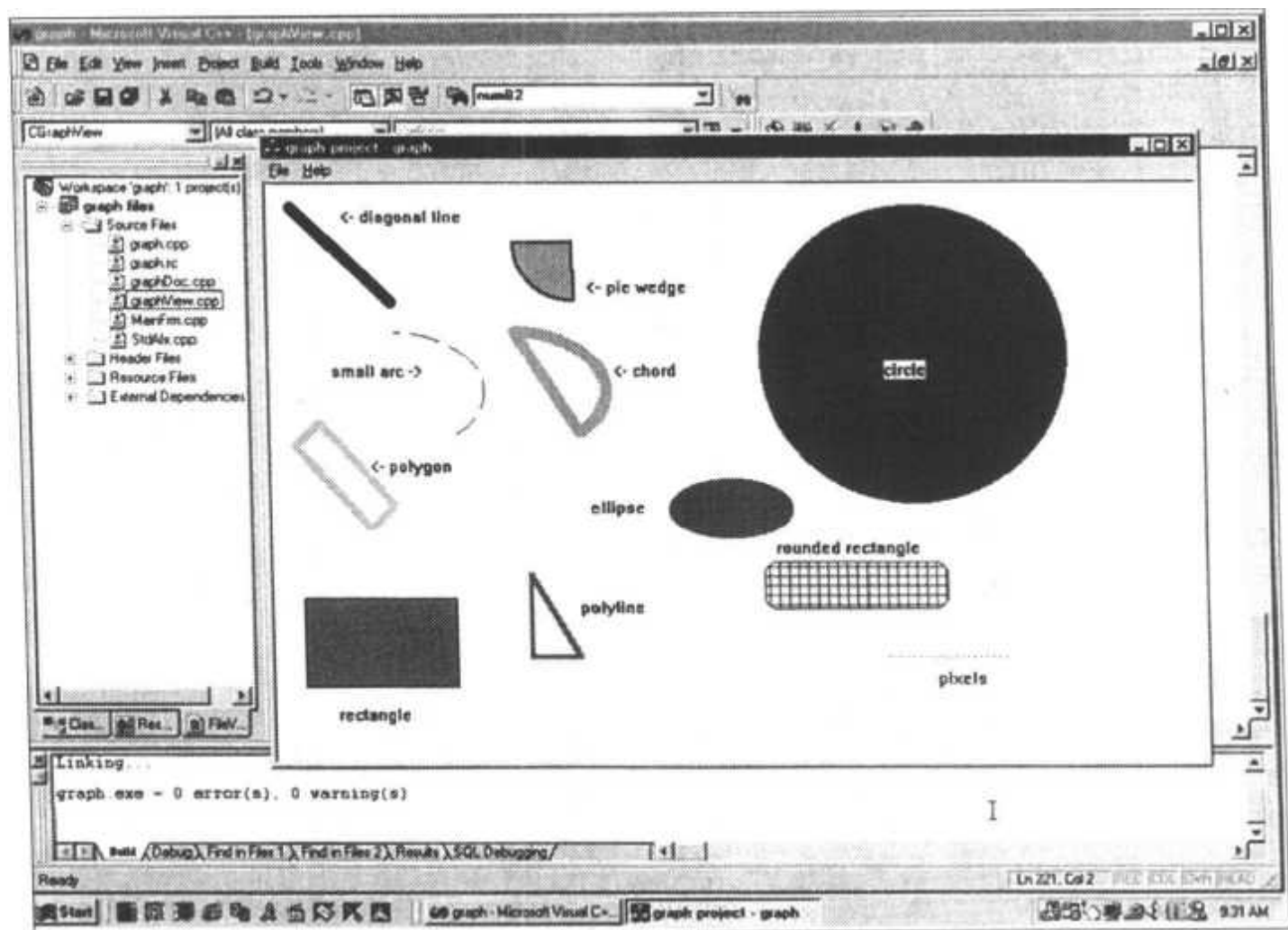


图 10-12 绘制客户区的各种形状的图形

10.5 剖面法

本章阐述的 `graph` 工程没有逻辑和语法错误。这章的目的是再熟悉 MFC 类库、AppWizard 和 ClassWizard，以及实现简单 GDI 图形函数的语法。

下面是完整的一节，阐述了另一个 Visual C++ 工具，该调试工具很有用，尤其是调试 MFC Windows 程序。或许我们曾经写过一些代码并问过这样的问题：“为什么我们的程序运行了这么长时间才完成？”，“我们编写的函数真的运行了否？”，“`LineTo()` 函数实际上执行了否？” 这些问题和更多问题的答案可以通过使用 Visual C++ 的剖面法 (profiling) 调试工具找到。

为了使剖面法对工程有效，要使用 **Project|Settings** 菜单项，然后选择 **Project Settings** 对话框中的 **Link** 标签，选择其中 **Enable profiling** 复选框，如图 10-13 所示。

在开始调试前，使用 **Build|Profile** 菜单项打开 **Profile** 对话框，如图 10-14 所示。

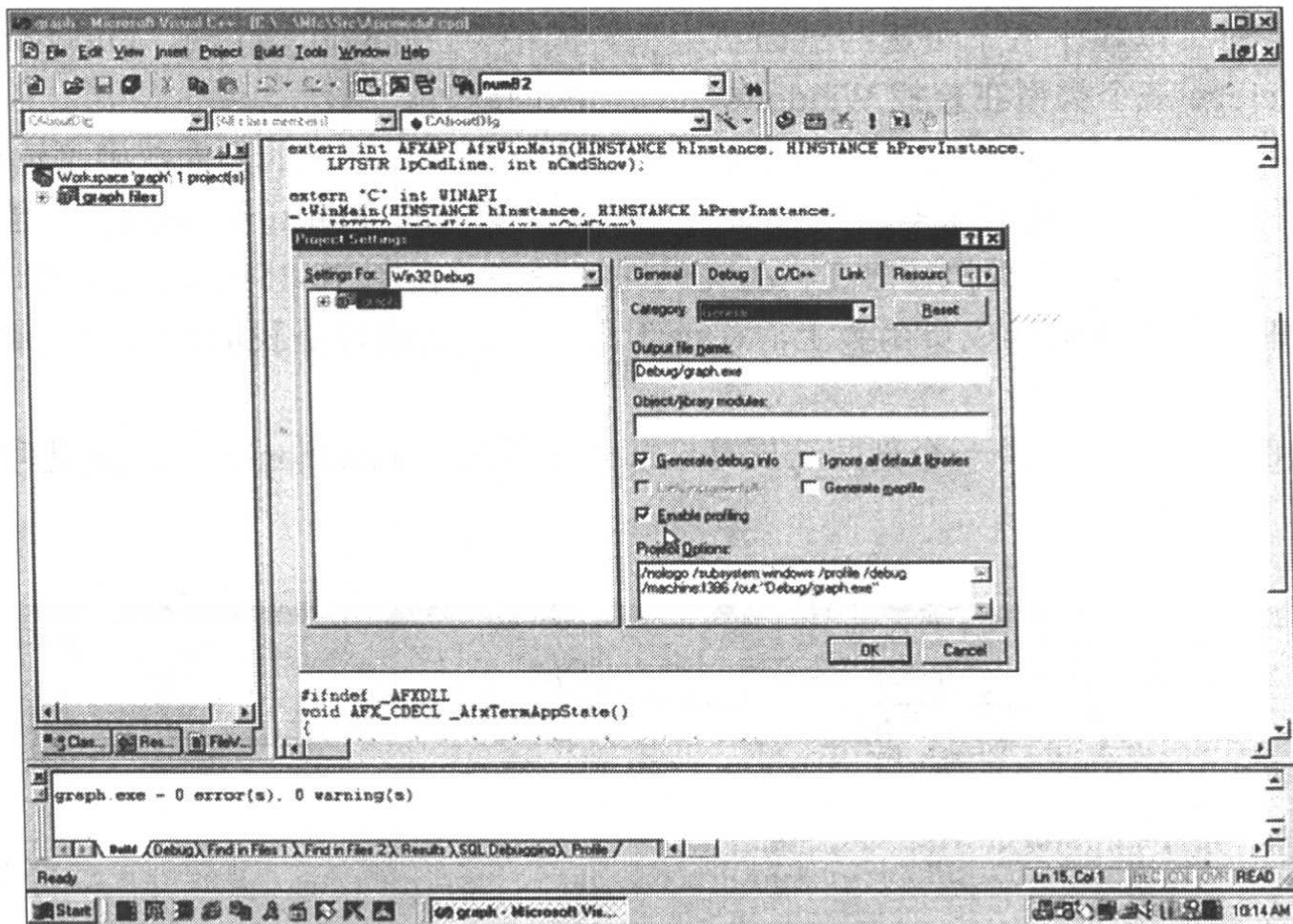


图 10-13 在 Project Settings 对话框中使剖面法生效

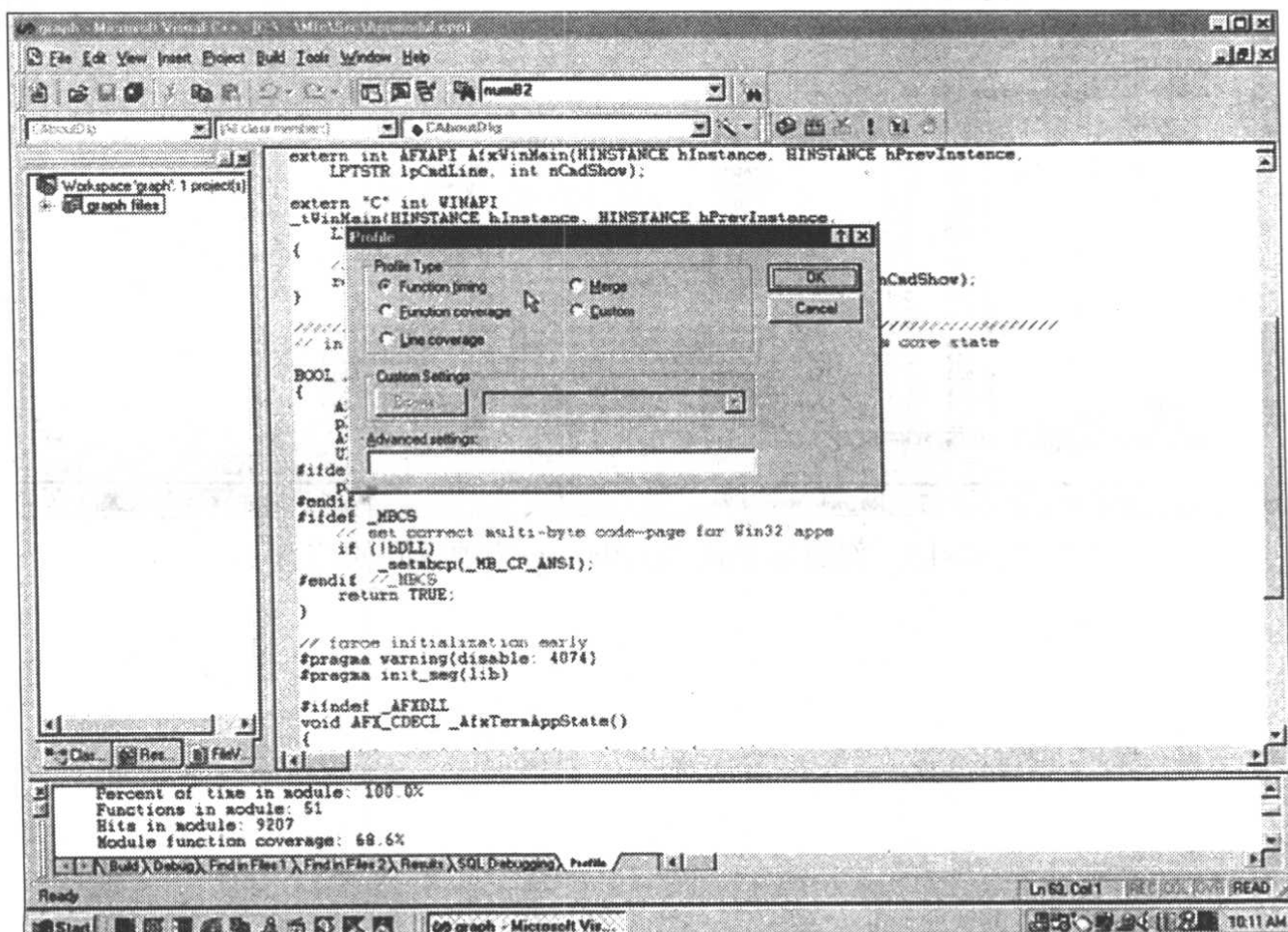


图 10-14 在 Profile 对话框中允许选择剖面法类型



Profile 对话框显示五个可选的剖面法类型。其中, Function timing、Function coverage 和 Line coverage 三个类型在查找 Windows 代码中的问题时有用。

图 10-15 显示了为 graph 工程选择 Function timing 类型后产生的剖面法屏幕输出。

如果程序执行中可能有某些时间方面的差错,剖面法的 Function timing 功能提供的信息,可以帮助我们查看工程执行时在何处花的时间最多。在图 10-15 中,大量的时间花在工程初始化和执行 OnPaint()成员函数上。在该工程中,此处或许是我们希望花最多时间之处。

而图 10-16 说明了另外一个剖面法屏幕,即当工程选择 Function coverage 选项后产生的屏幕。

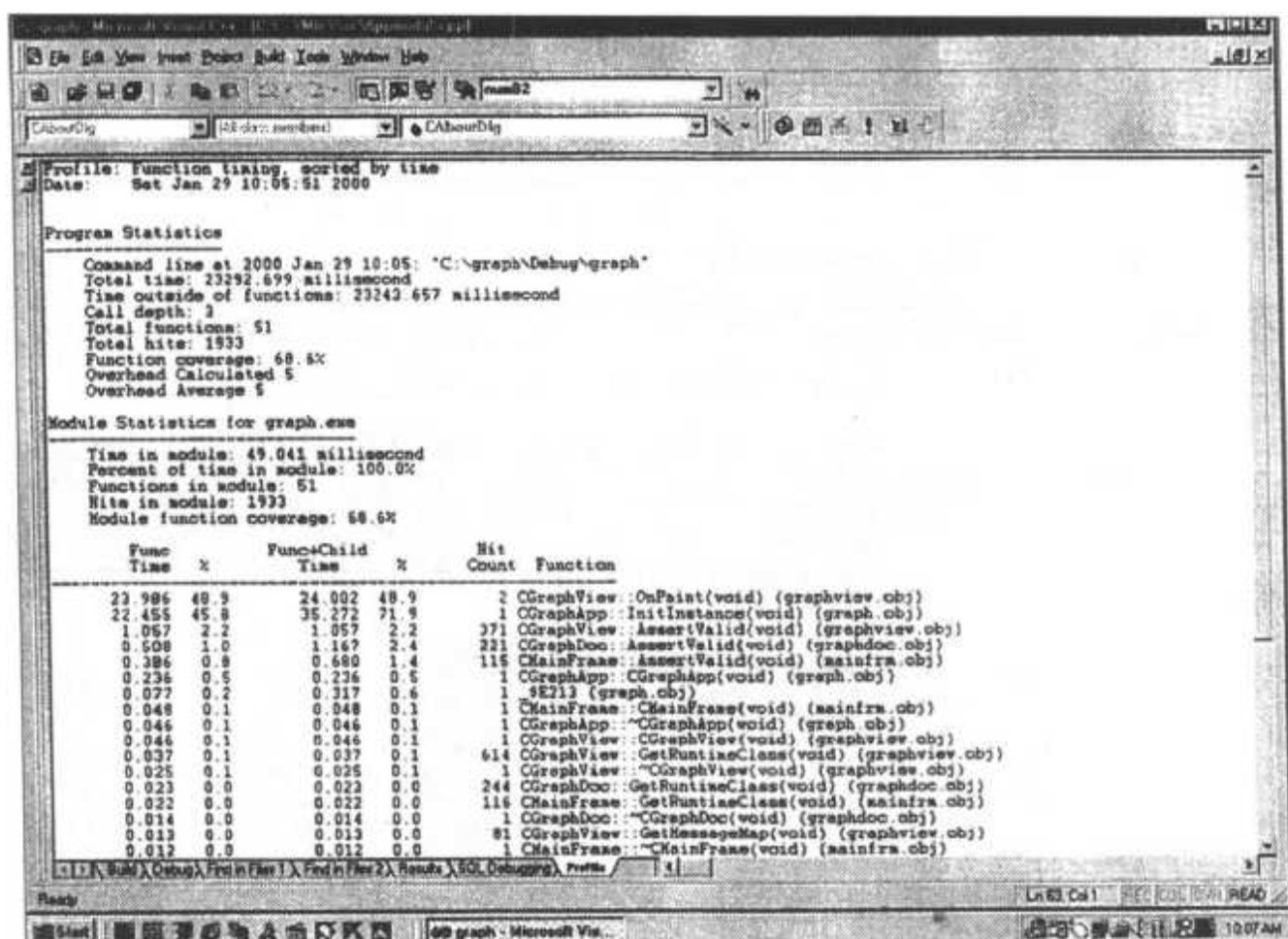


图 10-15 剖面法 Function timing 选项返回的信息

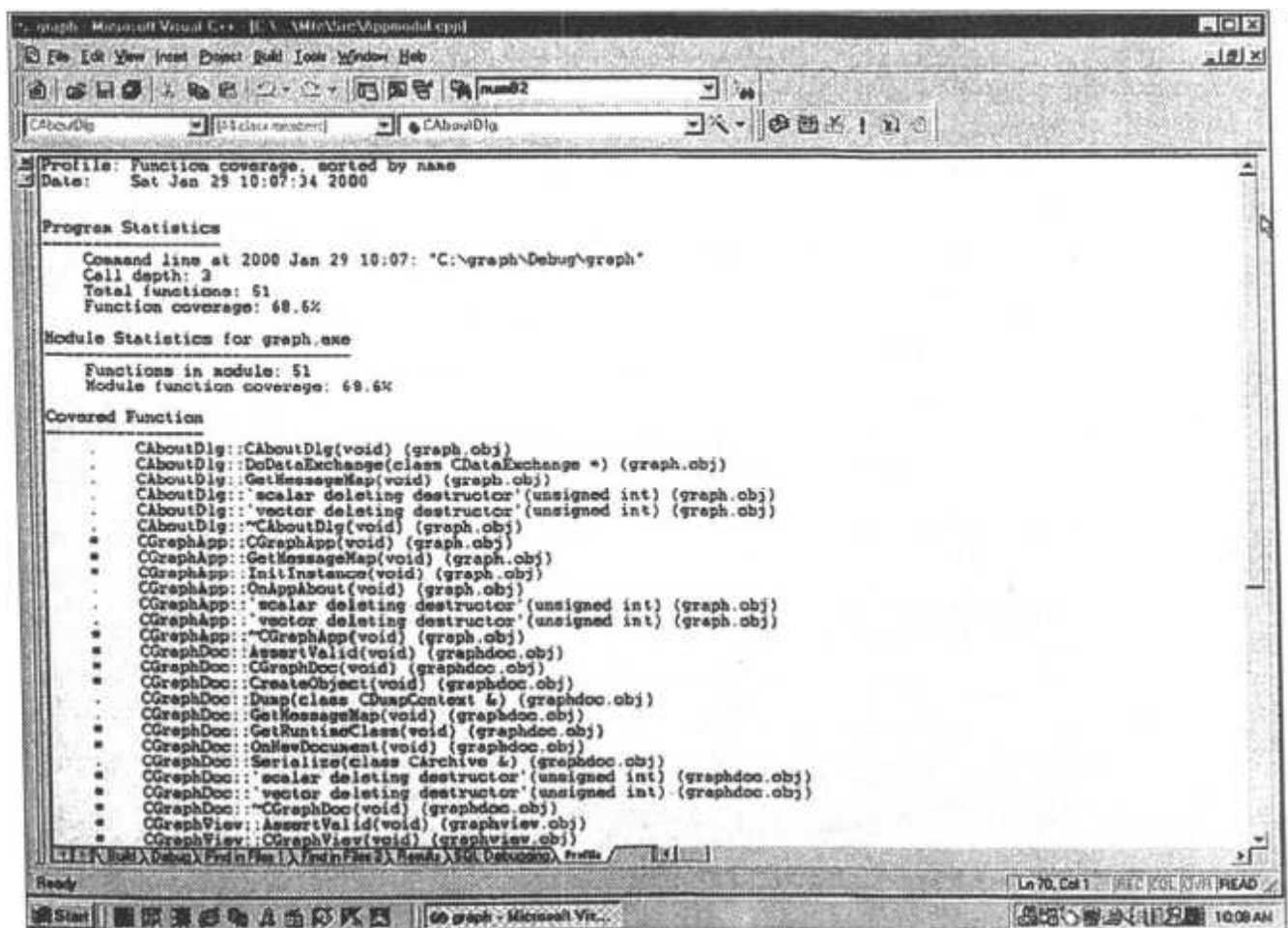


图 10-16 剖面法 Function coverage 选项返回的信息

该屏幕提供的信息显示了在工程运行期间使用的函数，每次工程运行并不调用所有的函数。例如，在这一 graph 工程的运行中，我们没有运行 About 对话框部分。但是程序的确执行了 InitInstance() 函数，正如在图 10-15 中所看到的那样。

在许多 MFC Windows 程序中，通常检测某个特定的函数是否执行是很困难的。这是由面向对象程序设计固有特性决定的，面向对象程序设计相对于面向过程的自上而下的设计方式。剖面法的 Function coverage 选项可以帮助检测在工程运行期间，特定的函数是否运行以及何时运行的。

Function coverage 选项在定位一些更细节的问题上是有用的，剖面法提供的 Line coverage 选项，可用来检测在工程执行过程中特定的代码行是否运行过。图 10-17 显示了 graph 工程初始的 Line coverage 屏幕输出。

在图 10-17 中，做了星号(*)标记的行都至少执行了一次。设想我们要知道特定的图形函数是否执行过，如果我们错误地计算了坐标，这一函数可能执行了，但在 Windows 的客户区不能显示。剖面法可以帮助我们找到答案，图 10-18 显示了这工程中调用的 GDI 图形函数部分的 Line coverage 信息。

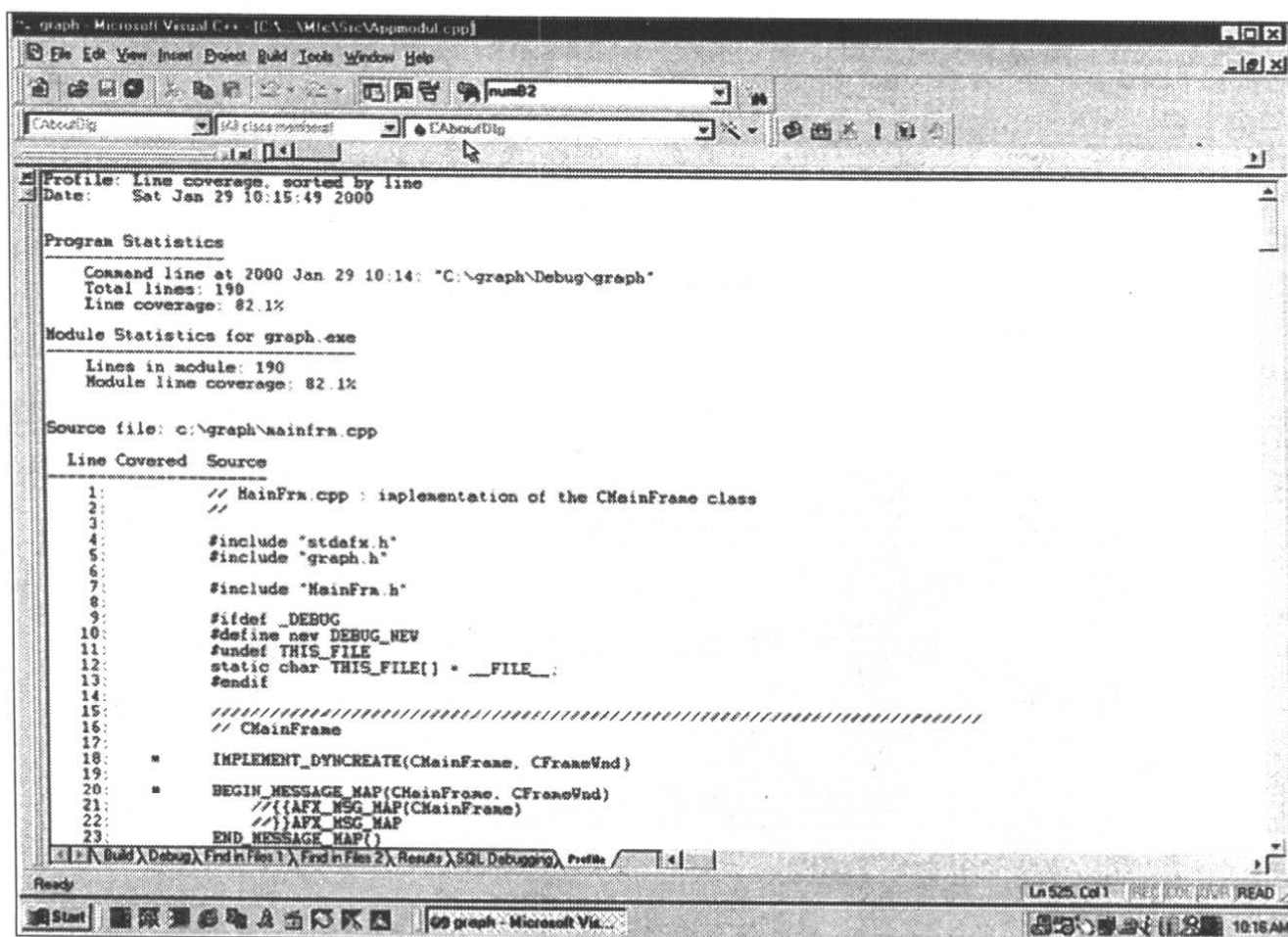


图 10-17 剖面法返回的 Line coverage 信息

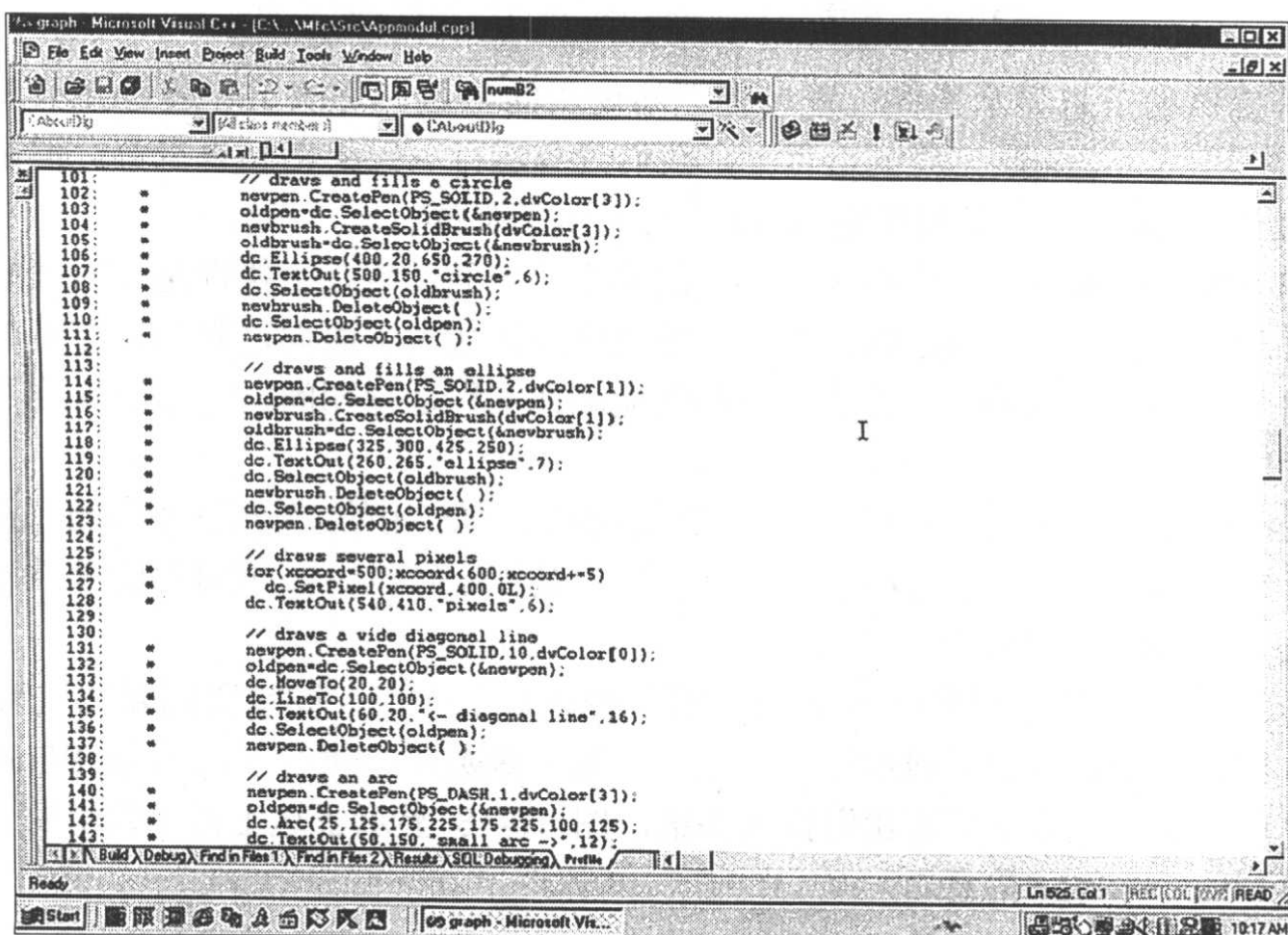


图 10-18 剖面法显示执行了所有的 GDI 图形函数

在后面的章节中，我们将看到，只要需要与时间和特定程序代码段的使用有关的程序运行信息，即可使用剖面法功能。

24x7

Microsoft 在有代表性地在 Visual Studio 的专业版和企业版中提供了剖面法功能。这将在以后的版本中有所变化。这是 Microsoft 最初对外的原则，后来根据 Visual C++ 程序员的需要做了修改。

1. 对于大的工程，可能不需要为整个程序用剖面法。要为工程的一部分用剖面法，可使用 PREP Phase I 命令，如/EXC 和/INC。例如，在许多情况下，通常不需要为由 AppWizard 产生的大部分代码用剖面法。
2. 剖面法的 Function timing 报告常常不准确。如果要求严格，则要运行几次程序，采用几次报告时间的平均值。对于多次运行得到的剖面法统计值，可以使用 Profiler 对话框的 Merge 选项合并。
3. 保持运行处理次数最少，这将产生更一致的时间统计(timing)结果。
4. 确定网络连接是断开的，避免有处理输入包的要求。
5. 如果 Function timing 和计算开销变化，则在 PREP Phase I 中使用/CB 开关。
6. 研究时间统计报告。如果有些函数报告花了大量时间，则可能是该函数调用剖面法子程序的时间加上该函数自己所用时间的结果。
7. 研究命中(hit)次数，这一函数被调用的次数正确否？被调用得过频繁否？
8. 即使在过去运行很好的一个函数也可能是个问题的来源，为工程中的所有函数查看其剖面法报告。
9. 记住函数的运行时间也包括函数的子函数。
10. 磁盘高速缓存的使用可以后面的剖面法更快的运行。
11. 用剖面法测定长函数中的问题是困难的，应将长函数分成更短的等价函数将帮助缩小问题的范围。
12. 不能将剖面法用于线程。但是通过使用/SF PREP(Phase I)选项，剖面法可以由启动线程的入口点的函数开始。

10.6 小结

本章除了介绍了 Microsoft Foundation Class 类库以外，还达到了许多目的。如果读者对使用 AppWizard 和 ClassWizard 建立应用程序熟悉，那么本章帮助快速回忆如何使用这些强有力的工具。本章还介绍了如何将各种 GDI 图形程序添加到利用 AppWizard 建立的模板代码中。还说明了面向过程函数和面向对象 MFC 类 Windows 平台之间的紧密联系。

Visual C++ profiler 最重要的优点是检测程序运行时间引起的问题、函数使用问题和指定代码行执行引起的问题。 ■

第 11 章

定位、分析和修复
MFC Windows 代码中的错误





第10章回顾了如何利用 Microsoft Foundation Class(MFC)类库以及 AppWizard 和 ClassWizard 等重要的工具开发工程。在第 10 章中,还学习了另一个工具 profiler,它能帮助我们发现 C++代码中的问题。

本章将使用各种调试工具定位、分析和修复 MFC Windows 程序中的问题。

我们在前面的章节中分析的所有编程问题,没有一个比有关内存分配和内存泄漏问题更难定位、分析和修复。在本章中,第一个工程就是考察这样一个问题。通过研究这一例子,将学习用于我们自己程序的一般技术。

本章中的第二个例子涉及到在窗口的客户区绘制傅立叶级数。这一例子中的错误不如前一个例子那样隐晦。但是,当检查出错误时,我们将看到其是普通的错误,甚至在不大复杂的程序中也容易被忽略。

11.1 内存问题

计算机的内存问题,即内存不足,在程序开发中,总是一个限制因素。看来计算机迫切需要越来越多的内存。在早期的计算机中,16KB 内存处理所有的程序。现在,256MB 或 512MB 内存看来才刚刚够用。Windows 系统使用复杂的内存管理程序帮助控制和优化内存使用,也包括磁盘缓存。当出现程序管理不当、分配超界或内存泄漏时,就发生严重错误。大部分 MFC 程序允许 Windows 系统管理分配内存资源,所以这部分通常没问题。然而,不是由系统分配处理的程序部分的内存泄漏是另外一回事。

内存分配和内存泄漏常被当成同样的问题,当在堆上分配了内存而一直不释放将发生内存泄漏,其含义是内存不释放无法重新使用。这种类型的问题是十分难测定的,因为有这种类型问题的程序起初看上去运行很好。发现一个程序有这种类型问题只是看到了问题的表面现象。要解决它,首先要定位和分析。

在下面几节中,我们将关注一个问题,这一问题最初是一些从事 MFC 工程的高年级学生带给我们的。

11.1.1 有问题的代码

一组学生正从事与简单的抛物运动有关的 MFC Windows 工程,下面的名为 MLeak 的工程保留了程序问题的基本部分,没有包含完成任务所需的其他代码。

要建立这一工程,可遵照第 10 章使用 MFC AppWizard 模板生成应用程序的大致步骤。命名这一工程名为 MLeak。

AppWizard 将为 Mleak 工程生成模板代码。此处只有 MLeakView.cpp 文件需要修改。下面的程序清单以粗体字显示了 AppWizard 代码的全部修改内容。

```
// MLeakView.cpp : implementation of the CMLeakView class
//
#include "stdafx.h"
#include "MLeak.h"
#include "MLeakDoc.h"
#include "MLeakView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////
// CMLeakView
IMPLEMENT_DYNCREATE(CMLeakView, CView)
BEGIN_MESSAGE_MAP(CMLeakView, CView)
   //{{AFX_MSG_MAP(CMLeakView)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
// CMLeakView construction/destruction
CMLeakView::CMLeakView()
{
}
CMLeakView::~CMLeakView()
{
}
BOOL CMLeakView::PreCreateWindow(CREATESTRUCT& cs)
{
    return CView::PreCreateWindow(cs);
}
////////////////////////////////////
// CMLeakView drawing
void CMLeakView::OnDraw(CDC* pDC)
{
    CMLeakDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    LOGFONT lf;
    CFont NFont;
    CFont* pOFont;
    // specify a logical font
    lf.lfHeight = 50;
    lf.lfWeight=FW_NORMAL;
    lf.lfEscapement=0;
```



```
lf.lfOrientation=0;
lf.lfItalic=false;
lf.lfUnderline = false;
lf.lfStrikeOut = false;
lf.lfCharSet=ANSI_CHARSET;
lf.lfPitchAndFamily=34; //Arial
NFont.CreateFontIndirect(&lf);
pOFont = PDC->SelectObject (&NFont);
pDC->TextOut(20, 200, "This program has memory problems");
DeleteObject(pOFont);
}
////////////////////////////////////
// CMLeakView diagnostics
#ifdef _DEBUG
void CMLeakView::AssertValid() const
{
    CView::AssertValid();
}
void CMLeakView::Dump(CDumpContext& dc) const
{
    CView::Dump (dc);
}
CMLeakDoc* CMLeakView::GetDocument() // non-debug ver. Inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMLeakDoc)));
    return (CMLeakDoc*)m_pDocument;
}
#endif // _DEBUG
```

通过选择 Build|Rebuild All 菜单创建此工程的可执行文件。运行该程序看见的窗口类似于图 11-1 所示。

初步地查看，这一程序看上去似乎运行正确，它建立了一个逻辑字体，TextOut()函数将文本显示在窗口的客户区上。这有什么问题？但是此程序有内存泄漏问题，此时还不能检测到，也许应该让程序再运行几分钟。实际上，程序的问题跟程序运行的时间长短没关系。用鼠标按住程序窗口的边框，执行窗口缩放操作 50 到 100 次，即可发现如图 11-2 显示的情形。

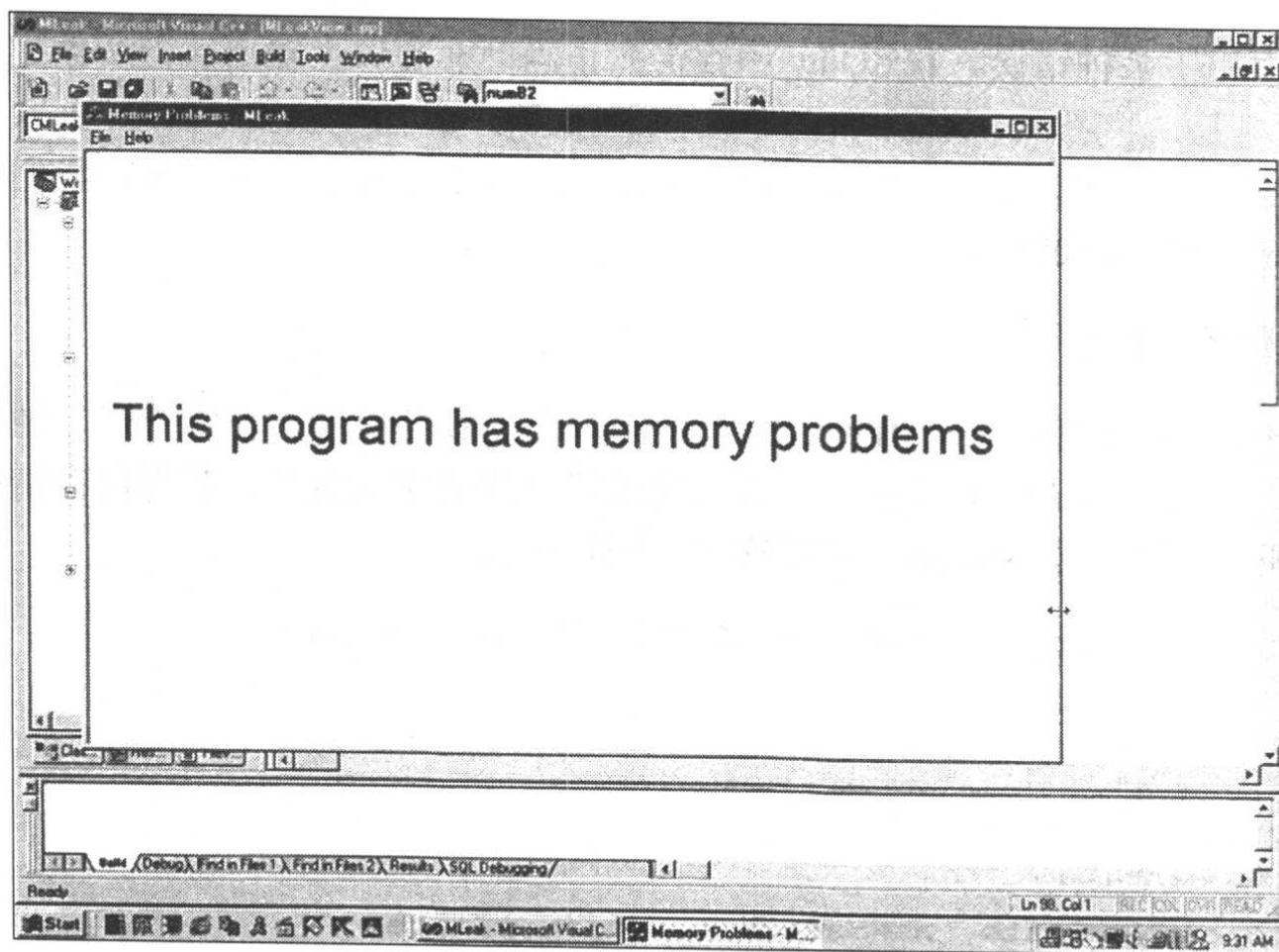


图 11-1 工程 MLeak 最初运行产生的窗口

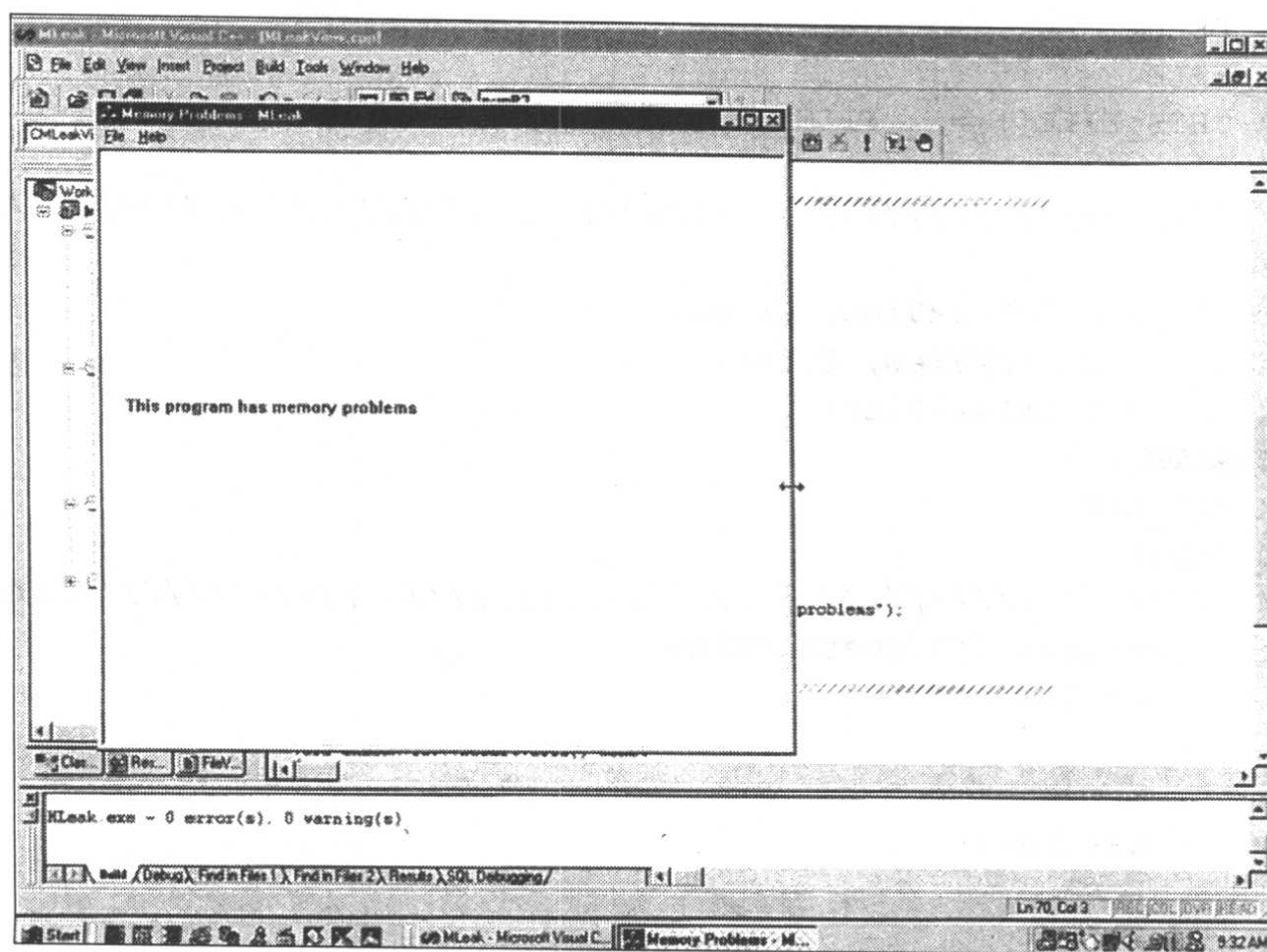


图 11-2 缩放若干次后的 MLeak 工程的窗口



在字体上出了某种错误。TextOut()函数仍然正确地在窗口上输出,但所创建的逻辑字体不正确了。

那组学生起初猜想问题在 OnDraw()方法的字体创建过程中。在下节,我们将看到他们定位和分析这一问题的过程。

11.1.2 定位和分析

学生们不得不定位和分析 MLeak 工程中的内存问题的根源。非常幸运,他们知道用来查找内存使用的几个 MFC 类和函数,适当地放置这些类和函数可以获得内存情况说明。通过适当的分析,可以确定内存是否被正确地分配和释放。

```
// MLeakView.cpp : implementation of the CMLeakView class
//
#include "stdafx.h"
#include "MLeak.h"
#include "MLeakDoc.h"
#include "MLeakView.h"
CMemoryState oldMemState, newMemState, diffMemState;
CFont NFont;
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////
// CMLeakView
IMPLEMENT_DYNCREATE(CMLeakView, CView)
BEGIN_MESSAGE_MAP(CMLeakView, CView)
    //{{AFX_MSG_MAP(CMLeakView)
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
// CMLeakView construction/destruction
CMLeakView::CMLeakView()
{
}
CMLeakView::~CMLeakView()
{
}
BOOL CMLeakView::PreCreateWindow(CREATESTRUCT& cs)
{

```



```

        return CView::PreCreateWindow(cs);
    }
    ///////////////////////////////////////////////////
    // CMLeakView drawing
    void CMLeakView::OnDraw(CDC* pDC)
    {
        CMLeakDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        LOGFONT lf;
        Cfont NFont;
        CFont* pOFont;
        lf.lfHeight = 50;
        lf.lfWeight = FW_NORMAL;
        lf.lfEscapement=0;
        lf.lfOrientation=0;
        lf.lfItalic= false;
        lf.lfUnderline = false;
        lf.lfStrikeOut = false;
        lf.lfCharSet=ANSI_CHARSET;
        lf.lfPitchAndFamily=34; //Arial
        NFont.CreateFontIndirect(&lf);
        pOFont = pDC->SelectObject(&NFont);
        pDC->TextOut(20, 200, pDoc->myCString);

        DeleteObject(pOFont);
#ifdef _DEBUG
        newMemState.Checkpoint();
        if(diffMemState.Difference(oldMemState,newMemState))
            TRACE("Difference between fire and now!\n\n");
            DiffMemState.DumpStatistics();
        }
#endif
    }
    ///////////////////////////////////////////////////
    // CMLeakView diagnostics
#ifdef _DEBUG
    void CMLeakView::AssertValid() const
    {
        CView::AssertValid();
    }
    void CMLeakView::Dump(CDumpContext& dc) const
    {
        CView::Dump(dc);
    }

```



```
}
CMLeakDoc* CMLeakView::GetDocument() // non-debug ver. inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMLeakDoc)));
    return (CMLeakDoc*)m_pDocument;
}
#endif // _DEBUG
int CMLeakView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
#ifdef _DEBUG
        oldMemState.Checkpoint();
#endif
    return 0;
}
```

应用程序初始化窗口时，为了获得对堆的统计，使用 ClassWizard 将 OnCreate()方法添加到 MLeakView.cpp 类中，这容易调用 oldMemState.Cheakpoint()。

在所有的字体设置工作完成后，执行 OnDraw()方法中的下列代码。

```
#ifdef _DEBUG
    newMemState.Checkpoint();
    if (diffMemState.Difference(oldMemState, newMemState)) {
        TRACE("Difference between first and now!\n\n");
        diffMemState.DumpStatistics();
    }
#endif
```

调用 newMemState.Checkpoint()抓取了堆(heap)的新的数据，调用 diffMemState.Difference()返回了初始化和当前值之间差别信息。统计值是调用 diffMemState.DumpStatistics()卸出的。因为这一信息包含在 OnDraw()方法中，并且 OnDraw 方法响应窗口缩放产生的 WM_PAINT 消息。所以每次缩放窗口，将打印统计值，这是我们最初碰到的问题。

下列是本程序在 Debugger 中运行时的简化输出：

```
Difference between first and now!
0 bytes in 0 Free Blocks.
-36 bytes in 0 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
```

```

Largest number used: 9 bytes.
Total allocations: 87 bytes.
Difference between first and now!
0 bytes in 0 Free Blocks.
-36 bytes in 0 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 9 bytes.
Total allocations: 132 bytes.
.
.
.
Diffence between first and now!
0 bytes in 0 Free Blocks.
-36 bytes in 0 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 9 bytes.
Total allocations: 14307 bytes.
Difference between first and now!
0 bytes in 0 Free Blocks.
-36 bytes in 0 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 9 bytes.
Total allocations: 14352 bytes.
    
```

如果仔细研究这一清单，将注意到卸出内容的只有一行有变化。那就是 Total allocations 行，看上去每次窗口缩放时这一行中的值都增加。更确切地说，每次增加 45 个字节。当我们考虑全部系统内存时，这一耗费似乎不算太大，但是查看最后卸出的统计数是 14352 字节，这一数字是缩放窗口相当多次后报告的！要记住的是，这一小的内存泄漏对程序产生的动态效果是其破坏了逻辑字体。

因为内存泄漏的如此少，学生们断定，是由于他们忘记为逻辑字体结构分配内存。所以，他们返回在 OnDraw() 方法中插入了如下一段粗体字代码：

```

LOGFONT lf;
CFont NFont;
CFont pOFont;
memset(&lf, 0, sizeof(LOGFONT));
    
```



```
lf.lfHeight=50;  
lf.lfWeight=FW_NORMAL;  
lf.lfEscapement=0;  
.  
.  
.
```

这种逻辑使教授为其学生们感到骄傲。学生们意识到卸出数据之间差为 45 个字节，逻辑字体结构可能也是这么大。但是，尽管这一假设很好，但其仍不能解决问题。分配内存的总数继续增加。

经过多次调试以及最后期限的临近，学生们不得不采取特别的措施。他们想尽了种种办法。他们继续下一步推理，或许每次窗口缩放时，在字体创建过程中发生了某件事，泄漏了一些内存。解决的方案是，将创建字体的过程从 OnDraw() 中移到 OnCreate() 函数中。使用这种方法，逻辑字体是在建立窗口时创建的，而不是在 OnDraw() 方法响应 WM_PAINT 消息时创建的。这一想法很有价值，它是传统方法中最好的程序设计形式——不持续分配和释放相同的资源。但是，这种方式也没有使问题得到解决，内存分配总数继续增加。同学们已经采取了所能采取的诊断工具，这时一个同学建议删除逻辑字体，使用系统字体查看将发生什么，由此 OnDraw() 被改成以下形式：

```
void CMLeakView::OnDraw(CDC* pDC)  
{  
    CMLeakDoc* pDoc = GetDocument();  
    ASSERT_VALID(pDoc);  
    pDC->TextOut(20, 200, "This program has memory problems");  
#ifdef _DEBUG  
    newMemState.Checkpoint();  
    if(diffMemState.Difference(oldMemState, newMemState)) {  
        TRACE("Difference between first and now!\n\n");  
        diffMemState.DumpStatistics();  
    }  
#endif  
}
```

测试这程序，猜猜将发生什么？此处是供考察的一段卸出统计数据。

```
Difference between first and now!  
0 bytes in 0 Free Blocks.  
-36 bytes in 0 Normal Blocks.  
0 bytes in 0 CRT Blocks.  
0 bytes in 0 Ignore Blocks.  
0 bytes in 0 Client Blocks.  
Largest number used: 9 bytes.  
Total allocations: 87 bytes.
```

```

Difference between first and now!
0 bytes in 0 Free Blocks.
-36 bytes in 0 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 9 bytes.
Total allocations: 132 bytes.
.
.
.

```

```

Diffeence between first and now!
0 bytes in 0 Free Blocks.
-36 bytes in 0 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 9 bytes.
Total allocations: 5352 bytes.
Difference between first and now!
0 bytes in 0 Free Blocks.
-36 bytes in 0 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 9 bytes.
Total allocations: 5397 bytes.

```

什么也没改变。每次卸出的内存分配总数之间仍差 45 个字节，那么问题在何处？我们知道 OnDraw() 方法中只剩如下一行代码了。

```
pDC->TextOut(20,200,"This program has memory problem");
```

注释这行代码，重新编译和运行，再检测一次。我们将看到每个卸出数据中的内存分配总数保持常数。实际上，重新放置逻辑字体代码，内存分配总数仍保持常数。

结论是，分配给字符串的内存，每次屏幕刷新时又分配了一次。显然，解决问题的方法是使该字符串脱离 OnDraw() 方法，正如同学们移出创建逻辑字体资源那样。在下节我们将看他们对这一问题的最终解决。

24x7 内存诊断使用的参数

要使用内存诊断工具，必须使诊断跟踪有效，它们是在 Debugger 中设置的。要使内存诊断有效或无效，Microsoft 建议调用全局函数 AfxEnableMemoryTracking()。这一函数使诊断内存分配器有效或无效。Debugger



自动设置这一功能为有效，用 `AfxEnableMemoryTracking()` 函数作为开关，可以将此功能设置为关。如此处理程序执行速度将提高，而且有一个完整的简化版诊断信息。

改变 MFC 的全局变量 `afxMemDF` 可使特定的内存诊断功能有效，这一变量可以使用的值如表 11-1 所示。

如果需要，表 11-1 显示的值可以用“或”操作组合使用。

表 11-1 `afxMemDF` 变量的值

变量值	描述
<code>AllocMemDF</code>	用于打开内存分配诊断。这是缺省设置
<code>CheckAlwaysMemDF</code>	每次内存分配或释放时，调用 <code>AfxCheckMemory()</code> 函数
<code>DelayFreeMemDF</code>	当调用删除或释放时，延迟释放内存，直至程序接束。总的效果是内存分配的数量尽可能最大

24x7 定位内存泄漏

Microsoft 已经提供了一个分级显示，它可以一步步查找内存泄漏。我们略微修改这一分级显示使其适合我们这一工程，步骤如下：

1. 实例化一个 `CMemoryState` 对象。在进入我们怀疑的代码前调用 `Checkpoint` 方法，得到初始的内存使用资料。
2. 实例化另一个 `CMemoryState` 对象，在程序执行我们怀疑的代码后调用 `Checkpoint` 方法，得到最终的内存使用资料。
3. 如果需要，再实例化一个 `CMemoryState` 对象，调用其 `Difference()` 方法。当调用这一方法时，使用前两个 `CMemoryState` 对象作为参数。有区别时，这一函数将返回一个非 0 值。这表明至少有一些内存块没有被释放。

下面是使用 `CMemoryState` 对象的一段代码：

```
// Declaration of CMemoryState variables
#ifdef _DEBUG
    CMemoryState oldMemState, newMemState, diffMemState;
    oldMemState.Checkpoint();
#endif

.
.
.
    (Code to be tested placed here)
.
.
.

#ifdef _DEBUG
    newMemState.Checkpoint();
    if(diffMemState.Difference(oldMemState, newMemState))
```

```
{
    TRACE("Memory Leaked Here:\n\n" );
}
#endif
```

用 `#ifdef _DEBUG` 和 `#endif` 括起这段代码，以确保其只能编译为 Win32 Debug 版本。

表 11-2 列出了 `CMemoryState` 对象的其他操作，包括一些简略的描述。

当两个 `CMemoryState` 对象的区别不能供足够多的信息时，可以使用此处的其他操作。

表 11-2 `CMemoryState` 对象的操作

操作	描述
Difference	用于找出两个使用了 <code>Checkpoint()</code> 方法的 <code>CMemoryState</code> 对象的区别
DumpAllObjectsSince	用于卸出当前所有调用 <code>checkpoint()</code> 方法的 <code>CMemoryState</code> 对象的汇总
DumpStatistics	用于打印对 <code>CMemoryState</code> 对象的内存分配统计。通常放在 <code>Checkpoint</code> 调用之后

24x7 卸出内存统计

`CMemoryState` 对象的成员函数可以用于卸出当前的统计值，或两个 `CMemoryState` 对象的区别。这两种技术都可以用来帮助定位有关堆内存释放问题。

下列代码使用了在前面 24x7 代码获得的信息，确定当前内存统计：

```
TRACE("Current Memory Picture:\n\n");
NewMemState.DumpStatistics();
```

原统计数据和新统计数据之间的区别也容易获得：

```
if( diffMemState.Difference(oldMemState,newMemState))
{
    TRACE("Memory Leaked Here:\n\n");
    diffMemState.DumpStatistics();
}
```

`diffMemState.DumpStatistics()` 调用的输出形式为：

```
0 bytes in 0 Free Blocks
2 bytes in 1 Object Blocks
50 bytes in 5 Non-Object Blocks
Largest number used: 76 bytes
Total allocations: 304 bytes
```

列表中的第一行指明已经释放的块数，这在变量 `afxMemDF` 被设置为 `delayFreeMemDF` (见表 11-1) 时发生。第二行用于指明还有多少个对象仍保留在堆中，第三行用于表明有多少个非对象块(用 `new` 分配的)在堆中，第四行指出程序在任何给定时间使用的最大内存，最后一行指出工程使用的总的内存量。其中任何一行的问题都表明有内存泄漏。



11.1.3 修复工程

从事这一工程的学生们充分利用了 Debugger 和检测内存泄漏的技术。而其最初的推理路线导致他们走了错误的道路，他们迅速调整，回到正确的道路上，并正确地定位和分析了这一问题。

解决这一问题的关键是，获取得每次窗口缩放调用的 OnDraw() 方法在窗口上输出的字符串。尽管这可以在文件 MLeakView.cpp 中成功地解决，而 AppWizard 专门为这种存储和内存分配创建了几个文件，它们是工程文件 MLeakDoc.h 和 MLeakDoc.cpp。

此处是文件 MLeakDoc.h 中的一部分，显示了对工程中使用的字符串的声明：

```
// MLeakDoc.h : interface of the CMLeakDoc class
//
///////////////////////////////////////////////////////////////////
#ifndef _AFX_MLEAKDOC_H__7D597DAA_8B14_11D3_A7DE_0080AE000001__INCLUDED_
#define _AFX_MLEAKDOC_H__7D597DAA_8B14_11D3_A7DE_0080AE000001__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class CMLeakDoc : public CDocument
{
protected: // create from serialization only
    CMLeakDoc();
    DECLARE_DYNCREATE(CMLeakDoc)
    CString myCString;
    .
    .
    .
}
```

这一字符串可以在 MLeakDoc.cpp 文件中的构造函数中找到，显示在下段清单中：

```
// MleakDoc.cpp : implementation of the CMLeakDoc class
//
#include "stdafx.h"
#include "MLeak.h"
#include "MLeakDoc.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```



```

////////////////////////////////////
// CMLeakDoc
IMPLEMENT_DYNCREATE(CMLeakDoc, Cdocument)
BEGIN_MESSAGE_MAP(CMLeakDoc, Cdocument)
    //{{AFX_MSG_MAP(CMLeakDoc)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
// CMLEAKDoc construction/destruction
CMLeakDoc::CMLeakDoc()
{
    myCString = "This program doesn't have a leak";
}
    
```

最后，列出修改好的 MLeakView.cpp 代码，它可以在发行(release)版中使用：

```

// MLeakView.cpp : implementation of the CMLeakView class
//
#include "stdafx.h"
#include "MLeak.h"
#include "MLeakDoc.h"
#include "MLeakView.h"
CFont NFont;
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////
// CMLeakView
IMPLEMENT_DYNCREATE(CMLeakView, CView)
BEGIN_MESSAGE_MAP(CMLeakView, CView)
    //{{AFX_MSG_MAP(CMLeakView)
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
// CMLeakView construction/destruction
CMLeakView::CMLeakView()
{
}
CMLeakView::~CMLeakView()
{
}
    
```



```
BOOL CMLeakView::PreCreateWindow(CREATESTRUCT& cs)
{
    return CView::PreCreateWindow(cs);
}
////////////////////////////////////
// CMLeakView drawing
void CMLeakView::OnDraw(CDC* pDC)
{
    CMLeakDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CFont* pOFont;
    pOFont = pDC->SelectObject(&NFont);
    pDC->TextOut(20, 200, pDoc->myCString);

    DeleteObject(pOFont);
}
////////////////////////////////////
// CMLeakView diagnostics
#ifdef _DEBUG
void CMLeakView::AssertValid() const
{
    CView::AssertValid();
}
void CMLeakView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}
CMLeakDoc* CMLeakView::GetDocument() // non-debug ver. inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMLeakDoc)));
    return (CMLeakDoc*)m_pDocument;
}
#endif // _DEBUG
int CMLeakView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
    LOGFONT lf;
    memset(&lf, 0, sizeof(LOGFONT));
    lf.lfHeight = 50;
    lf.lfWeight = FW_NORMAL;
```

```

lf.lfEscapement=0;
lf.lfOrientation=0;
lf.lfItalic=false;
lf.lfUnderline = false;
lf.lfStrikeOut = false;
lf.lfCharSet=ANSI_CHARSET;
lf.lfPitchAndFamily=34; //Arial
NFont.CreateFontIndirect(&lf);

return 0;
}
    
```

为建立这一工程，记着按如上修改文件 MLeakDoc.h、MLeakDoc.cpp 和 MLeakView.cpp。

设计提示

回忆一下第 2 章中围绕着字符串存储的一个优化技术。例如，如果编译器使用 /GF 或 /Gf 选项开关，而正好在程序中有下面的代码：

```

char *s = "Pooling will not work here";
char *t = "Pooling will not work here";
    
```

这一选择使编译器能够将同样的字符串复制为单独的一个副本放置到 .exe 文件中，因为它们被复制到一个单独的位置，所以使用这种选项编译的程序将更小。

但是，这不是这工程中字符串所遇到的问题。这不是多个字符串情况，而是单独的字符串被重复分配内存，这样 /GF 或 /Gf 开关对这一问题没有作用。

如果我们已经几次走过这条路，将很快弄清这是程序代码中的内存陷阱。但是对于第一次遇到它的人，是难以定位和检测的，所以这使我们更有经验。

11.2 绘图问题

许多电子工程和物理系的学生遇到的一个有趣的程序是计算傅立叶级数。法国数学家 Baron Jean Baptiste Joseph Fourier(1768-1830)发现几乎任何周期波形都可以通过简单地叠加正确的正弦谐波组合构造。傅立叶的结论产生了各种波形，从矩形波到三角波。电子工程师经常对再现音响设备中的矩形波感兴趣，因为矩形波是由基本的正弦波和相关的泛音组成，放大器和其他通信设备的质量依赖于如何可以很好地再现这些信号。我们将讨论傅立叶级数的组成，更详细的处理，可以在大学物理或电子工程课本中找到。

正规的傅立叶等式通常表示为下列形式：

$$y = A + A_1(\sin wt) + A_2(\sin 2wt) + A_3(\sin 3wt) + A_4(\sin 4wt) + A_5(\sin 5wt) \dots$$



一些周期波只包含奇次或偶次谐波项。另外一些则包括所有的项。在一些周期波中，相邻谐波项的符号正负交替。下面的例子通过将傅立叶级数中的奇次谐波项加在一起，构造了一个矩形波。傅立叶级数中的谐波项越多，最终的结果越接近精确的矩形波。对于矩形波，一般的傅立叶级数等式变为如下形式：

$$y \approx (\sin wt) + (1/3)(\sin 3wt) + (1/5)(\sin 5wt) + (1/7)(\sin 7wt) \dots$$

注意，只有奇次谐波对最终的结果有用。很显然，如果只有一个谐波项，结果将是正弦波。每个后继的高次谐波项乘以一个分数，换句话说，每个后继的谐波项对结果的影响越来越小。

程序分别计算傅立叶级数的每一项，并不断计算每个单项的和。这样当有 100 个谐波的傅立叶级数时，要计算 100 个正弦值，加在一起形成波形的单个点。但这必须逐个计算画在窗口的每个点。所以，100 次计算乘 400 个点=40000。想一下，用计算器多长时间可算完。

11.2.1 有问题的代码

这一工程的名字是 Fourier，是由一组学生按第 10 章概括的步骤，使用 AppWizard 创建的。对于 AppWizard 生成的基本代码，他们加了一个对话框用于数据输入，用一个通用颜色对话框选择填充颜色，当然还有其他一些程序代码。

并不是所有的 AppWizard 的文件都与本问题有密切关系，所以我们将只调查 FourierView.cpp 中的代码。

```
// FourierView.cpp : implementation of the CFourierView class
//
#include "stdafx.h"
#include "Fourier.h"
#include "FourierDoc.h"
#include "FourierView.h"
// additional header files needed
#include "FourierDlg.h"
#include "math.h"
CColorDialog dlg1; // include common color dialog box
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////
// CFourierView
IMPLEMENT_DYNCREATE(CFourierView, CView)
BEGIN_MESSAGE_MAP(CFourierView, CView)
```

```

//{{AFX_MSG_MAP(CFourierView)
ON_WM_SIZE()
ON_COMMAND(IDM_FOURIER, OnFourier)
ON_COMMAND(IDM_COLOR, OnColor)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
// CFourierView construction/destruction
CFourierView::CFourierView()
{
}
CFourierView::~CFourierView()
{
}
BOOL CFourierView::PreCreateWindow(CREATESTRUCT& cs)
{
    return CView::PreCreateWindow(cs);
}
////////////////////////////////////
// CFourierView drawing
void CFourierView::OnDraw(CDC* pDC)
{
    CFourierDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // all remaining code for Fourier Series
    int i,j,ang;
    double y, yp;
    CBrush newbrush;
    CBrush* oldbrush;
    CPen newpen;
    CPen* oldpen;
    // common color dialog box structure information
    // allow initial color value to be set
    dlg1.m_cc.Flags |= CC_FULLOPEN | CC_RGBINIT;
    dlg1.m_cc.rgbResult = pDoc->mycolor;
    pDC->SetMapMode(MM_ISOTROPIC);
    pDC->SetWindowExt(500,500);
    pDC->SetViewportExt(m_cxClient,-m_cyClient);
    pDC->SetViewportOrg(m_cxClient/20,m_cyClient/2);
    ang=0;
    yp=0.0;
    newpen.CreatePen(BS_SOLID,1,RGB(0,0,0));
    oldpen=pDC->SelectObject(&newpen);
}
    
```



```
// draw x & y coordinate axes
pDC->MoveTo(0,240);
pDC->LineTo(0,-240);
pDC->MoveTo(0,0);
pDC->LineTo(400,0);
pDC->MoveTo(0,0);
// draw actual Fourier waveform
for (i=0; i<=400; i++) {
    for (j=1; j<=pDoc->myterms; j++) {
        y=(250.0/((2.0*j)-1.0))*\
            sin(((j*2.0)-1.0)*\
                (ang*((360*22)/(180*400*7))));
        yp=yp+(int)y;
    }
    pDC->LineTo(i,yp);
    yp-=yp;
    ang++;
}
// create brush from common color dialog box selection
// for waveform fill
newbrush.CreateSolidBrush(pDoc->mycolor);
oldbrush=pDC->SelectObject(&newbrush);
pDC->ExtFloodFill(150,10,RGB(0,0,0),FLOODFILLBORDER);
pDC->ExtFloodFill(300,-10,RGB(0,0,0),FLOODFILLBORDER);
// delete brush objects
pDC->SelectObject(oldbrush);
newbrush.DeleteObject();
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CFourierView diagnostics
#ifdef _DEBUG
void CFourierView::AssertValid() const
{
    CView::AssertValid();
}
void CFourierView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}
CFourierDoc* CFourierView::GetDocument() // non-debug inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CFourierDoc)));
}
```

```

        return (CFourierDoc*)m_pDocument;
    }
#endif // _DEBUG
////////////////////////////////////
// CFourierView message handlers
void CFourierView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Add your message handler code here

    // WHM: added for sizing and scaling window
    m_cxClient = cx;
    m_cyClient = cy;
}
void CFourierView::OnFourier()
{
    // TODO: Add your command handler code here

    // WHM: added to process dialog information
    FourierDlg dlg (this);
    int result = dlg.DoModal();

    if(result==IDOK) {
        CFourierDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pDoc->myterms=dlg.m_terms;
        Invalidate();
    }
}
void CFourierView::OnColor()
{
    // TODO: Add your command handler code here
    // WHM: added to process common color
    // dialog box information

    int result = dlg1.DoModal();
    if(result==IDOK) {
        CFourierDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        // get returned color from dialog box
        pDoc->mycolor = dlg1.GetColor();
        InvalidateRect(NULL,TRUE);
        UpdateWindow();
    }
}

```



```
}  
}
```

这一程序允许用户在对话框中输入要画的波形的谐波数，程序然后画一个坐标和一个傅立叶波形。画刻度和绘制波形使用下列计算式：

```
// draw actual Fourier waveform  
for (i=0; i<=400; i++) {  
    for (j=1; j<=pDoc->myterms; j++) {  
        y=(250.0/((2.0*j)-1.0))*\  
            sin(((j*2.0)-1.0)*\  
                (ang*((360*22)/(180*400*7))));  
        yp=yp+(int)y;  
    }  
    pDC->LineTo(i,yp);  
}
```

上述段代码在坐标轴上绘制了 400 个点。每个点是由 1 到 `nterms` 次谐波的计算值合成，`nterms` 变量中存放要计算的傅立叶级数的谐波数。每一项的重复变量是 `y`，计算的累加和保存在变量 `yp` 中，代表了那个横坐标对应的傅立叶级数点。画一条从当前点到新点的线使用的是 `LineTo()` 函数。表达式的 $((360*22)/(180*400*7))$ 部分将 360 度等分为 400 个点 $(360/400)$ ，并将度转换为弧度 $(\pi/180)$ 。

程序允许用户从一个通用颜色对话框为波形选择某种填充颜色，下列函数的功能是填充每个波峰。

```
pDC->ExtFloodFill(150,10,RGB(0,0,0),FLOODFILLBORDER);  
pDC->ExtFloodFill(300,-10,RGB(0,0,0),FLOODFILLBORDER);
```

在用户输入前，初始绘画需要绘制四个谐波项。图 11-3 显示了工程执行后的窗口。

看了图 11-3 后，我们断定，同学们远未取得满意的结果。程序何处出了错误？第一眼看上去，得到的结论是，波形没有画出来，波形的填充颜色涂满了整个屏幕。好的信息是画出了两条轴线。至少我们知道，`OnDraw()` 方法起作用了。如果不出现这种情况，可以使用 `profiler` 确认程序实际调用了哪些方法。

11.2.2 定位和分析

这将是检验调试技巧的好机会，检查程序和我们至今所做的分析给出的所有信息。要猜一猜问题在何处否？

要知道已经正确画出了两个坐标轴，我们不必担心 `OnDraw()` 方法的绘制功能。因为各种 `LineTo()` 函数的行为是正确的，所以同样其他 GDI 画图程序也是正确的。

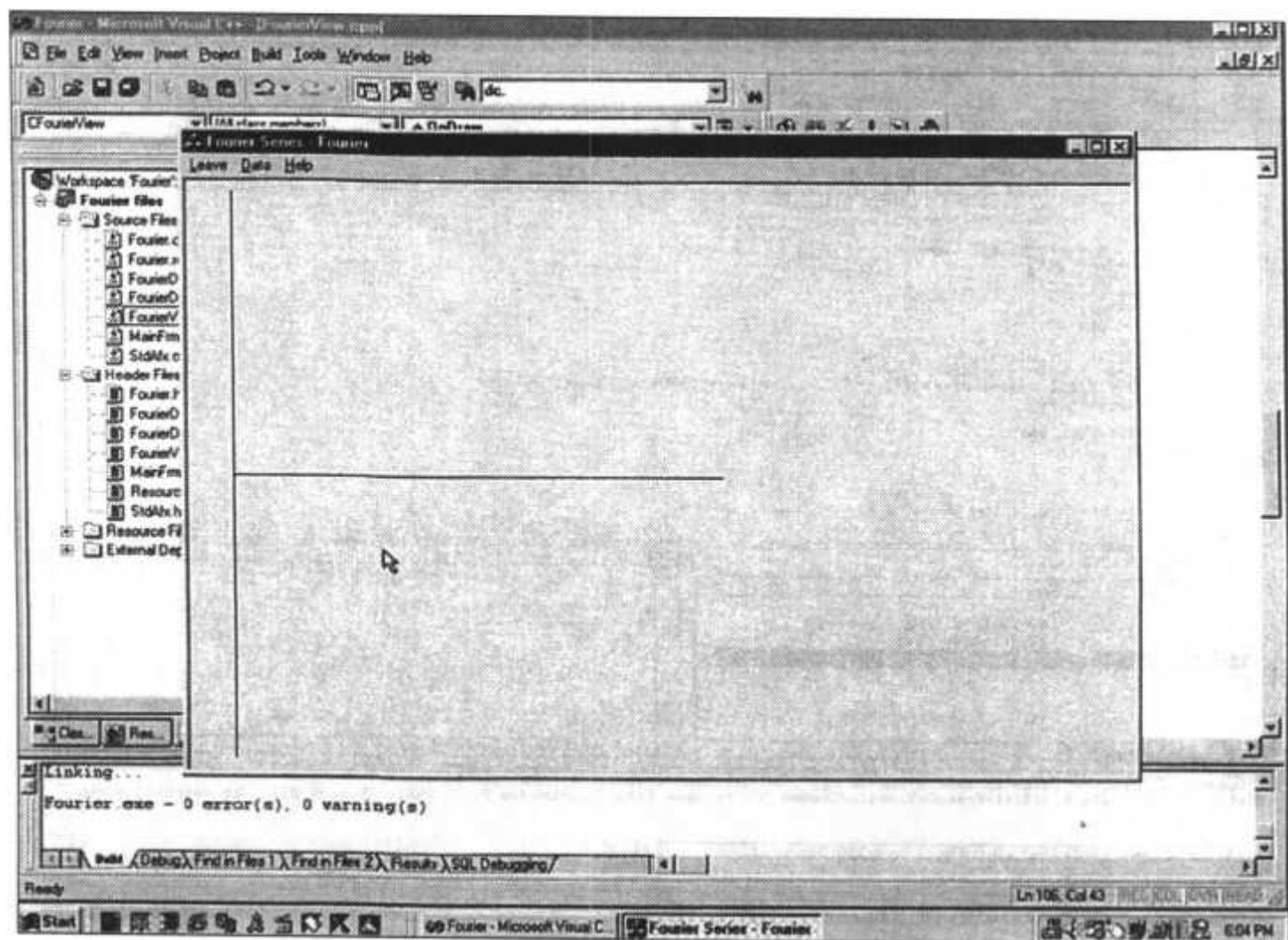


图 11-3 Fourier 工程起初的窗口

11.2.2.1 研究第一个问题

因为坐标轴画得正确，所以我们将注意力集中在计算傅立叶项的表达式上。这一表达式是什么？

检查一下内层循环：

```
y = (250.0 / ((2.0 * j) - 1.0)) * sin(((j * 2.0) - 1.0) * (ang * ((360 * 22) / (180 * 400 * 7))));
```

要分析这一问题，考虑这样一种情况：如果用户只选择一个傅立叶项，这一内层循环将简化为下面的算式：

```
y = 250 * sin(ang * 0.015707)
```

现在，计算式表达为一个正弦波，正弦波的值很容易计算(使用计算器)，在 Debugger 的 Watch 窗口跟踪，查看发生了什么。

使用 Debugger 的 Go(F5)键快速跳过一些度数，查看图 11-4，看到我们的结果是大约 40 度(45*360/400)。

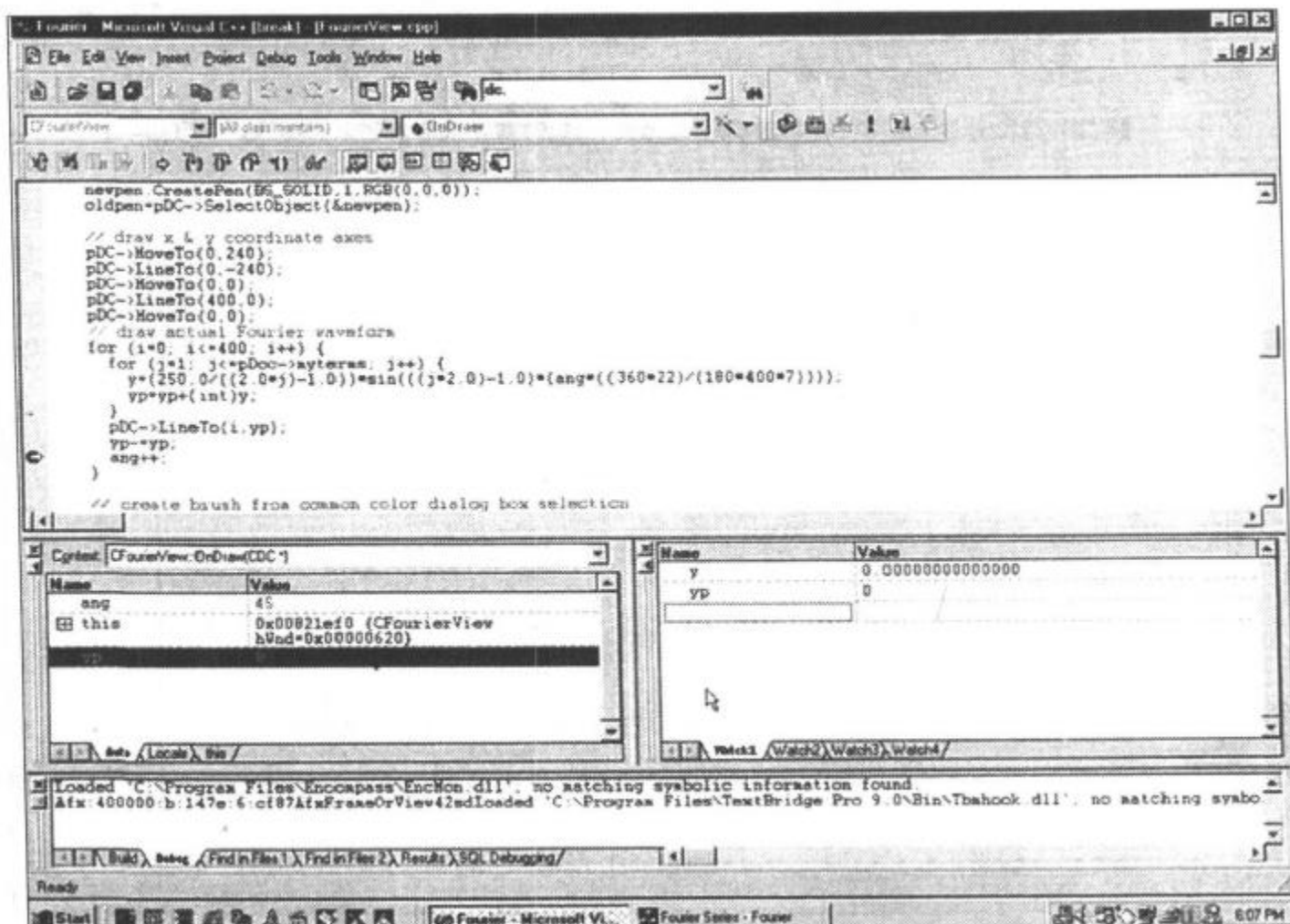


图 11-4 在 Watch 窗口中 yp 的结果始终为 0

好了，我们发现了问题的根源，y 值一直是 0，没有改变。y 没有变化，图就画不出来！你知道错误的根源否？问题在于使用下式计算单位值：

$$\text{Scaling value} = ((360 * 22)) / ((180 * 400 * 7))$$

用计算器，算出这一值是 0.1571428...。但是，在这一表达式中使用，整型值将返回结果为 0。这一值必须被转变为实型数。此外，这一表达式整理一下。我们知道 360.0 被 180.0 除是 2.0。pi 可以表示为 3.14159，所以我们将那部分代码替换为：

$$y = (250.0 / ((2.0 * j) - 1.0)) * \sin(((j * 2.0) - 1.0) * (\text{ang} * 2.0 * 3.14159 / 400.0));$$

通过 Debugger 再次运行程序，在 Watch 窗口中观察 y 变量。图 11-5 显示 40 度时修改算式后的 y 值。

这种情况，40 度的正弦值是 0.64...，这一值乘以 250 得出的值约为 160...。

要查看绘制的图是否正确，在 Debugger 外运行工程。将看到绘制的图形类似于图 11-6。

波形看上去类似预想的正弦波，因此，看来同学们已经改正了问题。接下来检验 4 个谐波绘制的图，如图 11-7 所示。

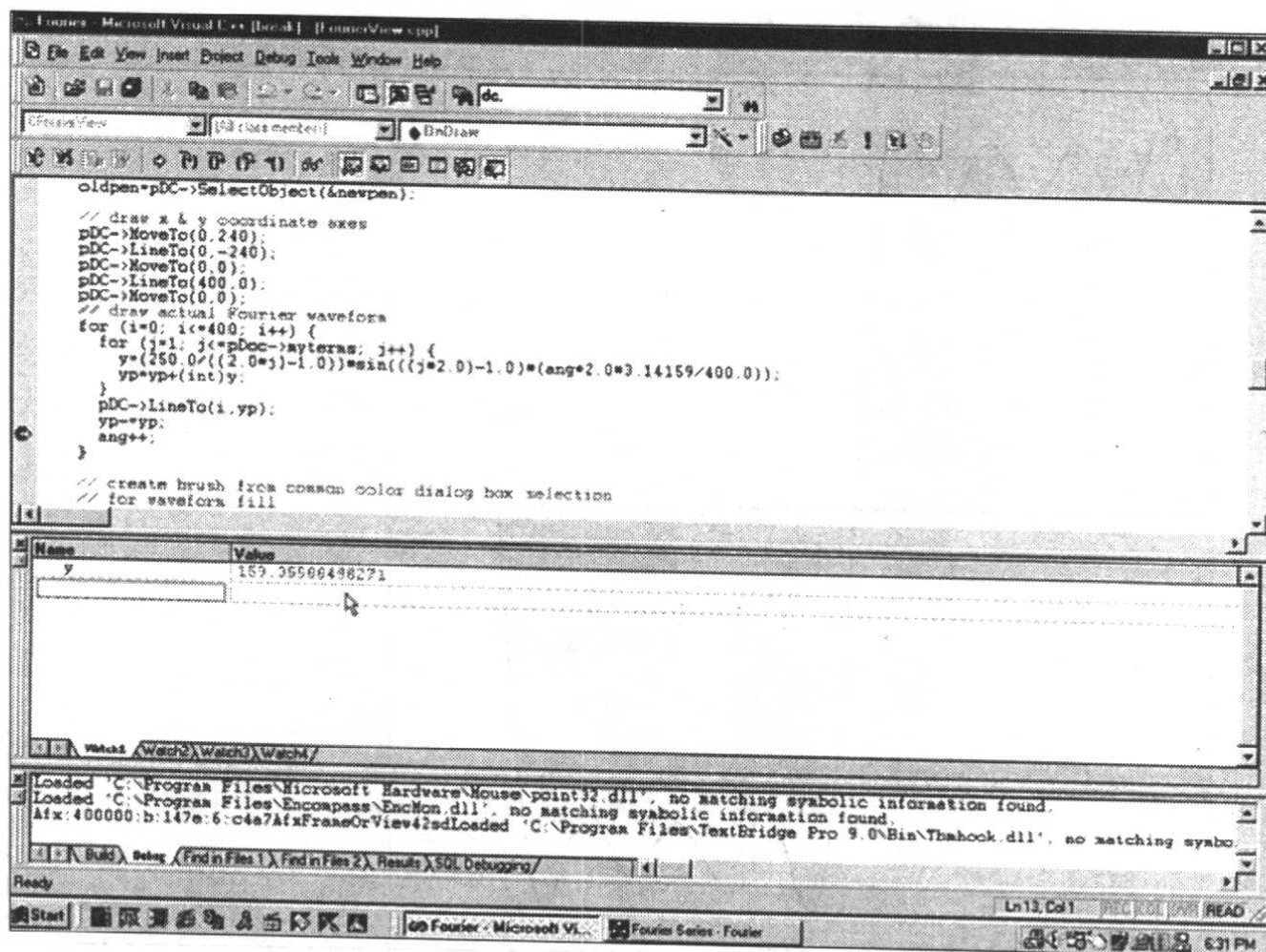


图 11-5 40 度时 y 的返回结果

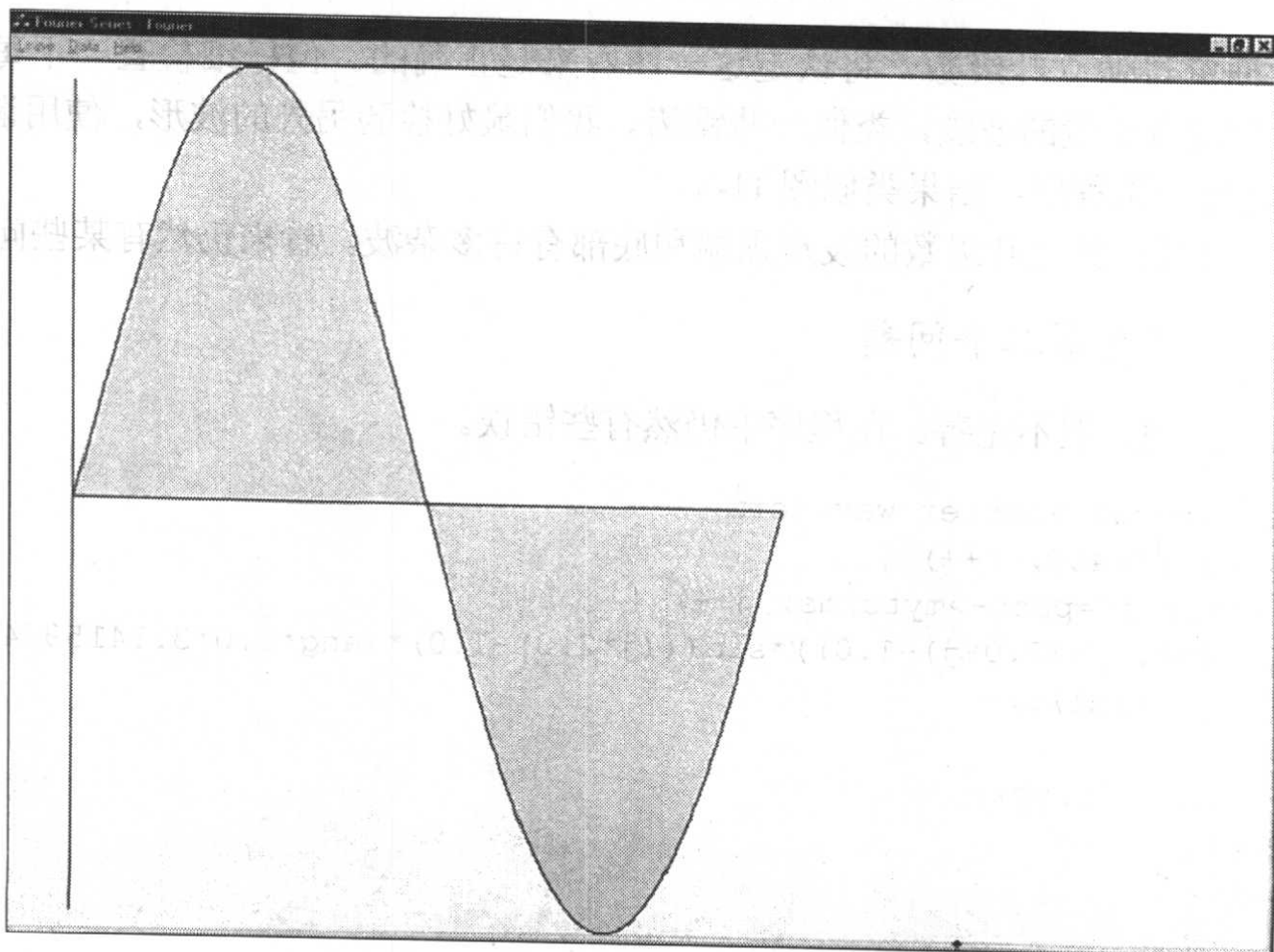


图 11-6 绘制只含一个谐波项的傅立叶级数

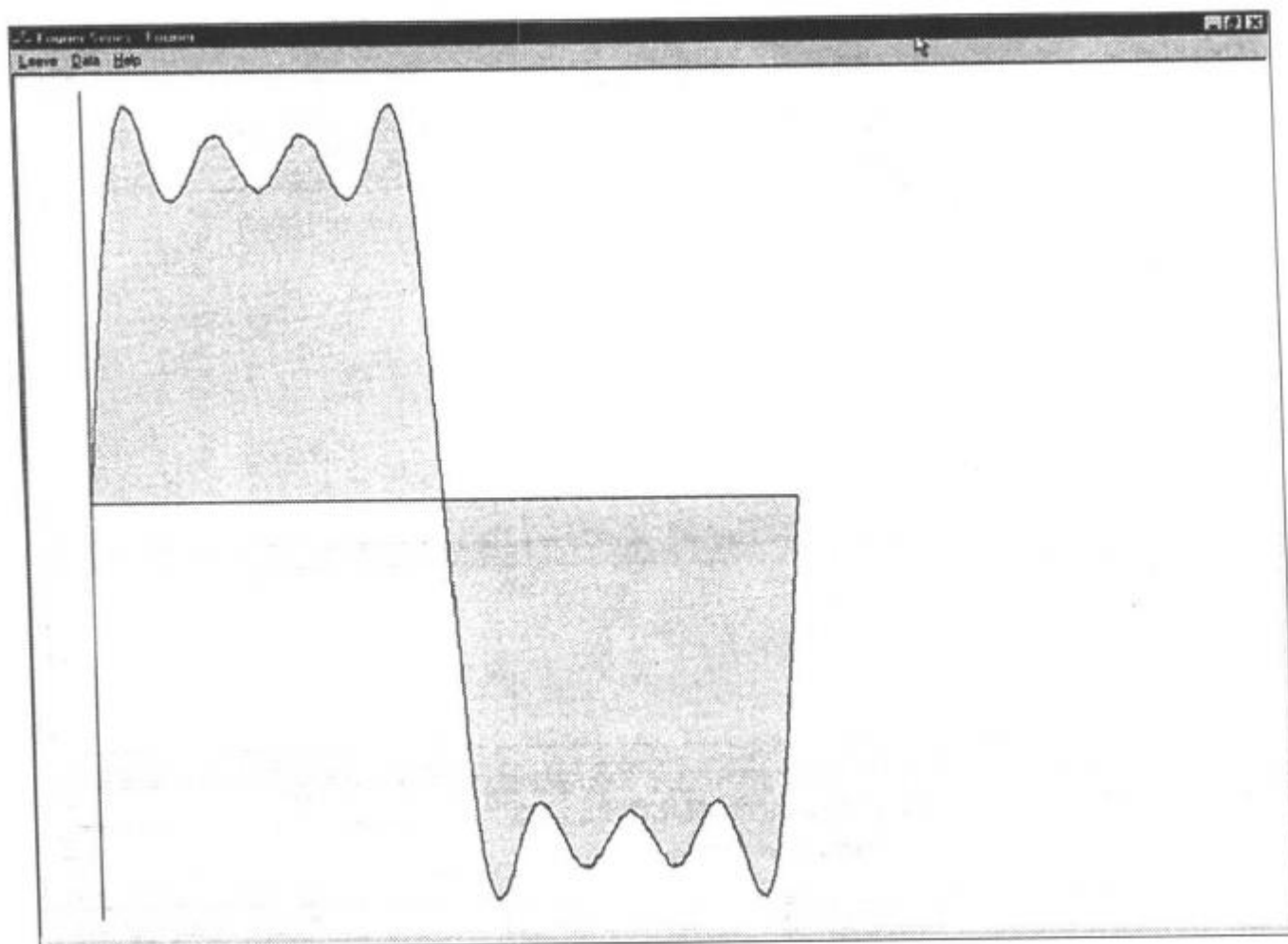


图 11-7 绘制含有 4 个谐波项的傅立叶级数

如果你研究过傅立叶级数，将认定这一图看来是正确的。但仔细检查一下每个波形顶端和底部，发现有少量的波纹，类似一些锯齿。我们最好检验另外的波形，使用谐波项数为 400 个，再运行一次程序，结果类似图 11-8。

400 个谐波项的傅立叶级数的波形顶端和底部有许多杂波。看来仍然有某些问题。

11.2.2.2 研究第二个问题

图形接近正确，但不完善。在程序中仍然有些错误。

```
// draw actual Fourier waveform
for (i=0; i<=400; i++) {
    for (j=1; j<=pDoc->myterms; j++) {
        y=(250.0/((2.0*j)-1.0))*sin(((j*2.0)-1.0)*(ang*2.0*3.14159/400.0));
        yp=yp+(int)y;
    }
    pDC->LineTo(i,yp);
    yp-=yp;
    ang++;
}
```

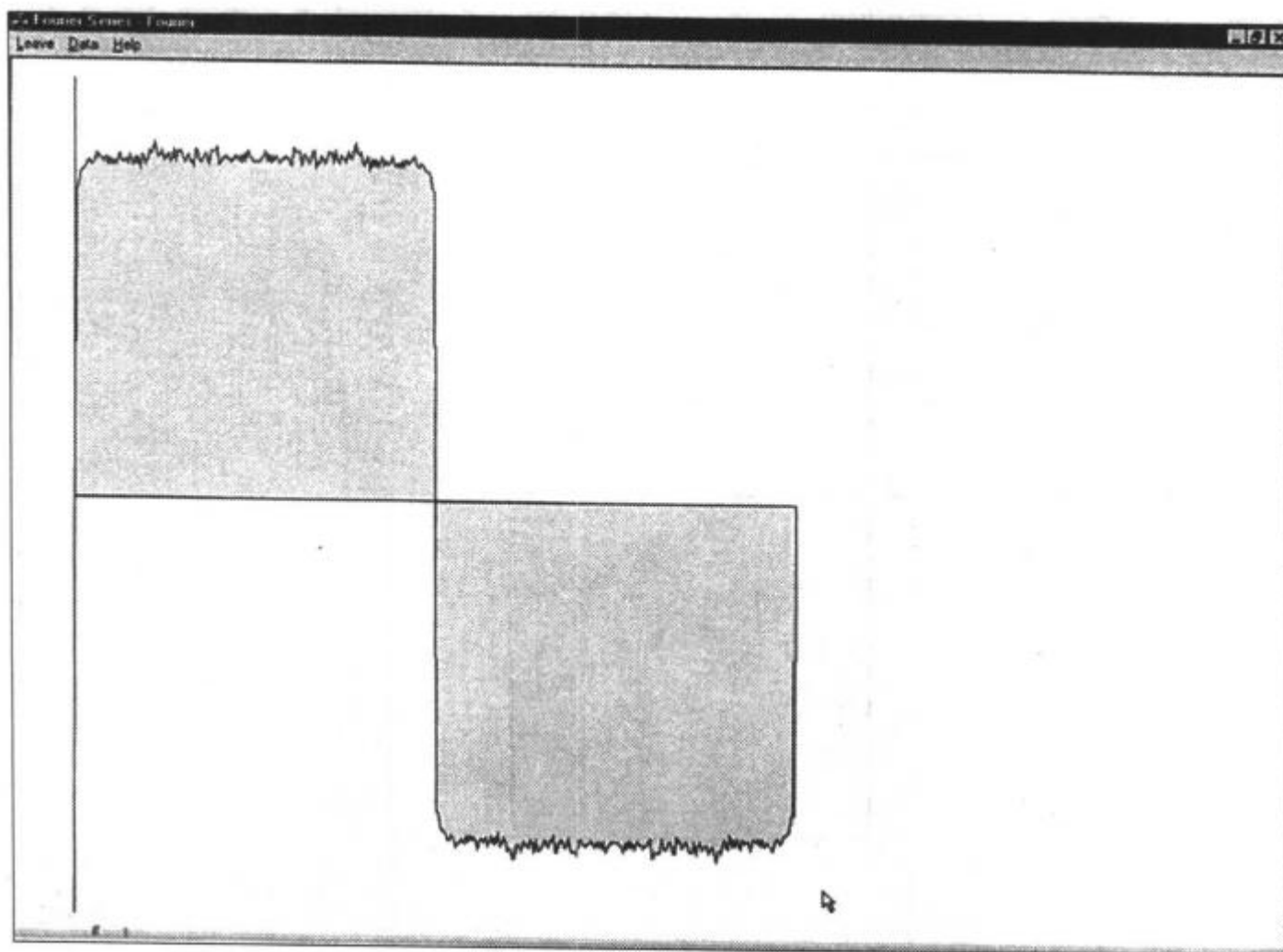


图 11-8 含 400 个谐波项的傅立叶级数的顶端和底部不太好

再检验表达式，问题的解决方法几乎就在眼前。确实，`LineTo()`函数需要一个整型值作为水平和垂直位置。这些值由 `i` 和 `yp` 变量提供。`y` 是双精度，在累加到 `yp` 变量之前已经转化为整型数。我们猜想，同学们对 `yp` 值的四舍五入执行的太早了。很容易改正。简单地声明 `yp` 为双精度，改变计算为下列方式：

```
// draw actual Fourier waveform
for (i=0; i<=400; i++) {
    for (j=1; j<=pDoc->myterms; j++) {
        y=(250.0/((2.0*j)-1.0))*sin(((j*2.0)-1.0)*(ang*2.0*3.14159/400.0));
        yp=yp+(int)y;
    }
    pDC->LineTo(i, (int)yp);
    yp-=yp;
    ang++;
}
```

通过在 `LineTo()`函数中转换 `double` 为 `int`，保持了最高的绘图精度，图 11-9 显示了修改为 400 个谐波后程序执行的结果。

余下的事就是用其他各种傅立叶级数项数检验程序。

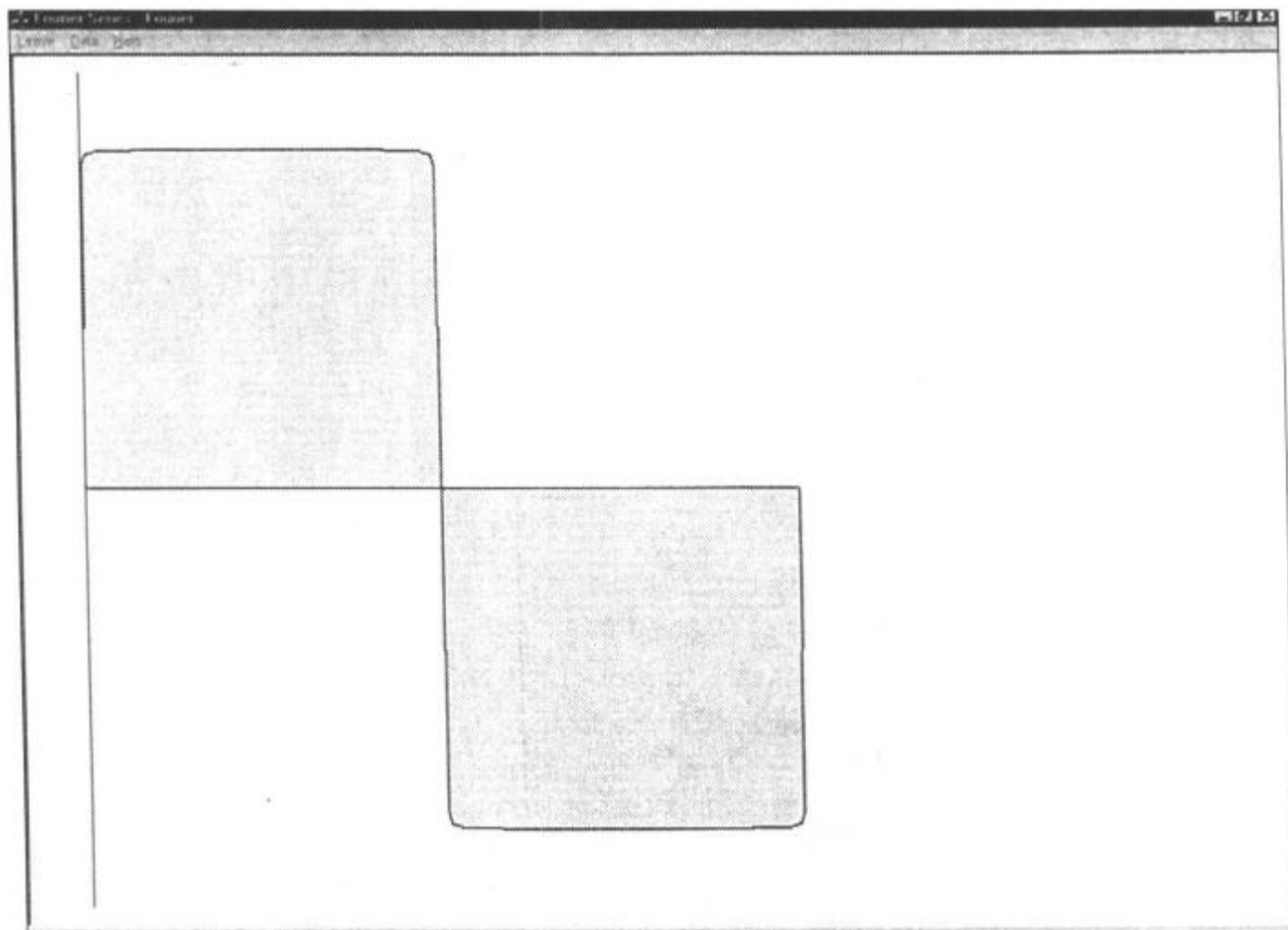


图 11-9 消除舍入误差后，大大地改进了绘出的图形

直到我们达到了 754 个谐波项，所有的改进完全起到了作用。对于比这一值大的所有值，波形绘制也是正确的，但整个波形是以绘制颜色填充的，如图 11-10 所示。

因为应用程序绝大多数时间工作正常，所以下面回到绘图问题上来。

11.2.2.3 研究第三个问题

我们不会责备学生将这仍然当成某种舍入问题。事实上，程序已经很好了，但其产生的结果与学生和指导老师最初想的相比有些出入。

我们的忠告是，回去，用 Debugger 的 Watch 窗口检查绘制到屏幕上的点值。单步通过 $754 \times 400 = 301600$ 点，真是点的问题。碰巧，一直到最后一个点值才发现问题。最后一个点的 y 值没有返回到 x 轴上封闭图形，但是 ExtFloodFill() 函数填充依赖于封闭的图形。

```
pDC->ExtFloodFill(150,10,RGB(0,0,0),FLOODFILLBORDER);  
pDC->ExtFloodFill(300,-10,RGB(0,0,0),FLOODFILLBORDER);
```

换句话说，ExtFloodFill() 函数在波形上发现了一个漏洞，当其填到最后部分，颜色漏到整个屏幕上。

学生们采用增加 pi (从 3.14159 到 3.14159265359) 的精度解决了这一程序的问题，图 11-11 显示当计算式中使用 100000 个谐波项时窗口的情况。

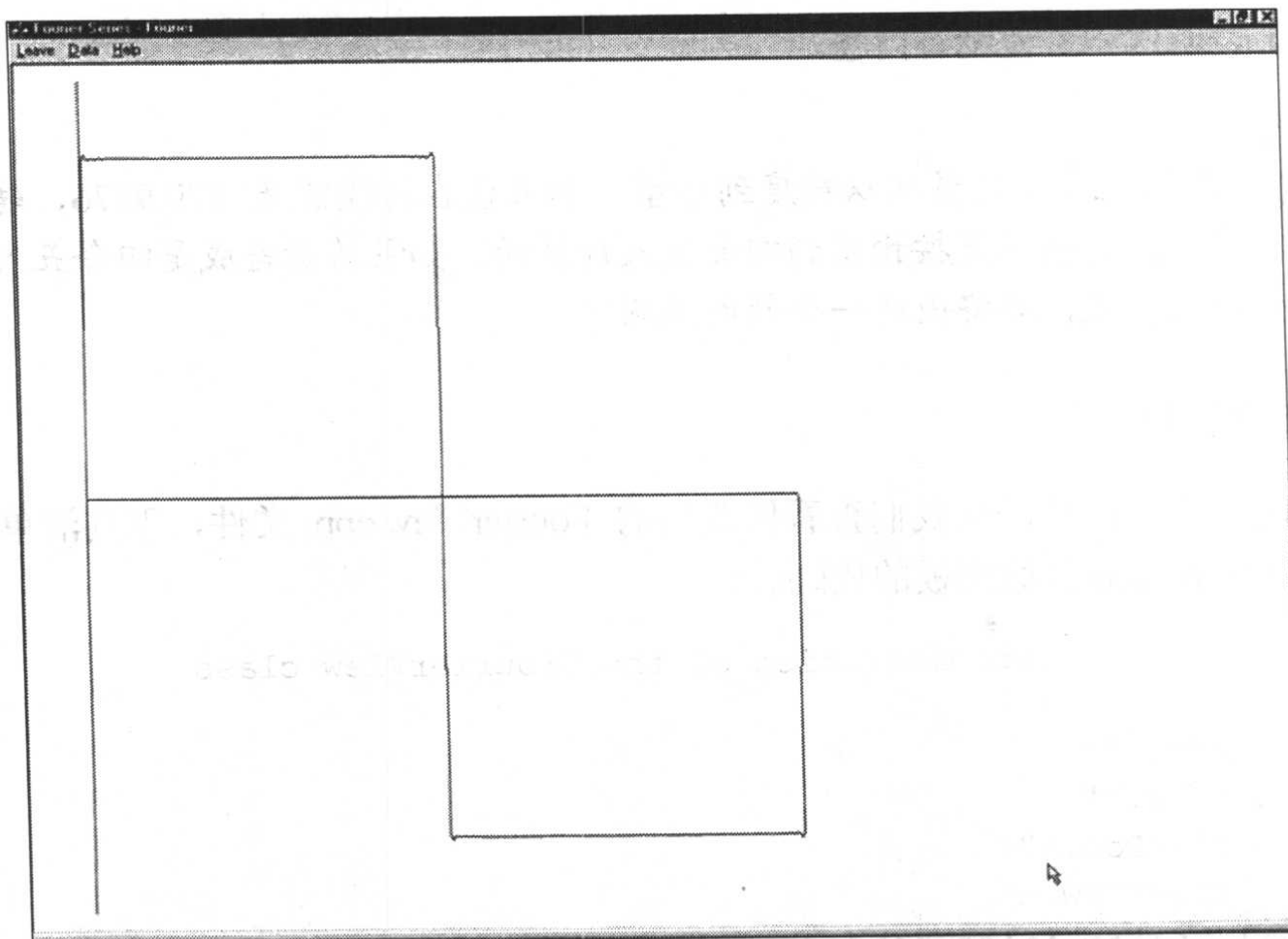


图 11-10 对于 754 项和更多项的傅立叶级数，填充颜色填满了整个屏幕

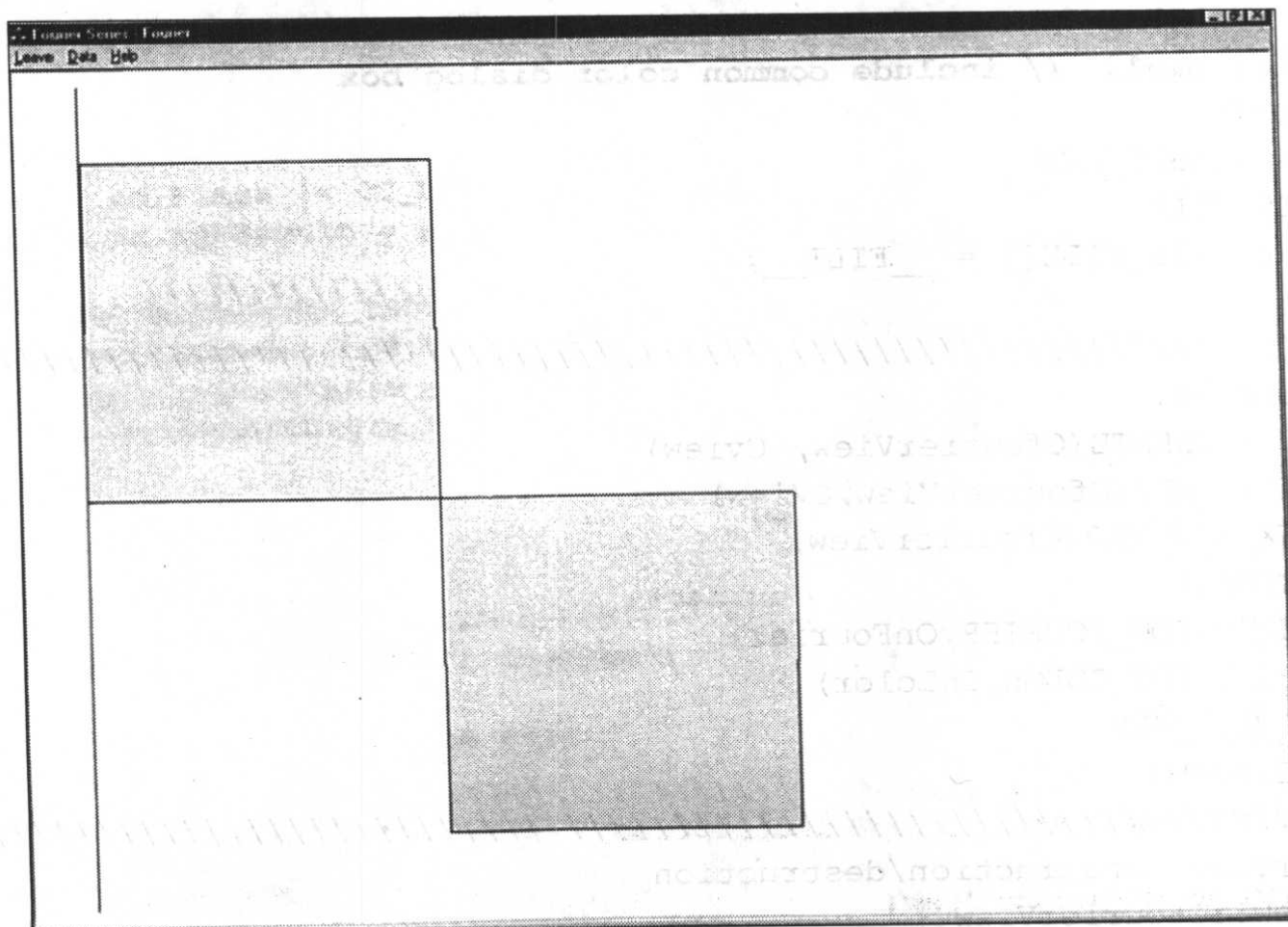


图 11-11 对于 100,000 项的傅立叶级数，得到一个好的图形



图 11-11 表明这一工程没有问题。

错误监视

LineTo()函数转化 yp 变量从双精度到整型。如果值在转换前是 179.9876, 转化后将变成 179。要观察的点的值不是按预想的四舍五入计算的, 如果将其当成是四舍五入, 以为封闭了图形以供颜色填充, 那将出现一个颜色漏洞。

11.2.3 修改工程

所有代码准备就位后, 让我们查看修改后的 FourierView.cpp 文件, 下列清单以粗体字显示了加到 AppWizard 初始模板的代码。

```
// FourierView.cpp : implementation of the CFourierView class
//
#include "stdafx.h"
#include "Fourier.h"
#include "FourierDoc.h"
#include "FourierView.h"
// additional header files needed
#include "FourierDlg.h"
#include "math.h"
CColorDialog dlg1; // include common color dialog box
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////
// CFourierView
IMPLEMENT_DYNCREATE(CfourierView, Cview)
BEGIN_MESSAGE_MAP(CfourierView, CView)
    // {{AFX_XSG_MAP(CfourierView)
    ON_WM_SIZE()
    ON_COMMAND(IDM_FOURIER, OnFourier)
    ON_COMMAND(IDM_COLOR, OnColor)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
// CFourierView construction/destruction
CFourierView::CFourierView()
{
}
```



```

CFourierView::~CFourierView()
{
}

BOOL CFourierView::PreCreateWindow(CREATESTRUCT& cs)
{
    return CView::PreCreateWindow(cs);
}

////////////////////////////////////
// CFourierView drawing
void CFourierView::OnDraw(CDC* pDC)
{
    CFourierDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // all remaining code for Fourier Series
    int i,j,ang;
    double y, yp;
    CBrush newbrush;
    CBrush* oldbrush;
    CPen newpen;
    CPen* oldpen;
    // common color dialog box structure information
    // allow initial color value to be set
    dlg1.m_cc.Flags |= CC_FULLOPEN | CC_RGBINIT;
    dlg1.m_cc.rgbResult = pDoc->mycolor;
    pDC->SetMapMode(MM_ISOTROPIC);
    pDC->SetWindowExt(500,500);
    pDC->SetViewportExt(m_cxClient,-m_cyClient);
    pDC->SetViewportOrg(m_cxClient/20,m_cyClient/2);
    ang=0;
    yp=0.0;
    newpen.CreatePen(BS_SOLID,1,RGB(0,0,0));
    oldpen=pDC->SelectObject(&newpen);

    // draw x & y coordinate axes
    pDC->MoveTo(0,240);
    pDC->LineTo(0,-240);
    pDC->MoveTo(0,0);
    pDC->LineTo(400,0);
    pDC->MoveTo(0,0);
    // draw actual Fourier waveform
    for (i=0; i<=400; i++) {
        for (j=1; j<=pDoc->myterms; j++) {
            y=(250.0/((2.0*j)-1.0))*sin(((j*2.0)-1.0)*(ang*2.0*3.14159265359/400.0));
        }
    }
}

```



```
        yp=yp+y;
    }
    pDC->LineTo(i,(int) yp);
    yp-=yp;
    ang++;
}
// create brush from common color dialog box selection
// for waveform fill
newbrush.CreateSolidBrush(pDoc->mycolor);
oldbrush=pDC->SelectObject(&newbrush);
pDC->ExtFloodFill(150,10,RGB(0,0,0),FLOODFILLBORDER);
pDC->ExtFloodFill(300,-10,RGB(0,0,0),FLOODFILLBORDER);
// delete brush objects
pDC->SelectObject(oldbrush);
newbrush.DeleteObject();
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CFourierView diagnostics
#ifdef _DEBUG
void CFourierView::AssertValid() const
{
    CView::AssertValid();
}
void CFourierView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}
CFourierDoc* CFourierView::GetDocument() // non-debug inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CFourierDoc)));
    return (CFourierDoc*)m_pDocument;
}
#endif // _DEBUG
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CFourierView message handlers
void CFourierView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Add your message handler code here

    // WHM: added for sizing and scaling window
    m_cxClient = cx;
    m_cyClient = cy;
```

```

}
void CFourierView::OnFourier()
{
    // TODO: Add your command handler code here

    // added to process dialog information
    FourierDlg dlg (this);
    int result = dlg.DoModal();

    if(result==IDOK) {
        CFourierDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pDoc->myterms=dlg.m_terms;
        Invalidate();
    }
}
void CFourierView::OnColor()
{
    // TODO: Add your command handler code here
    // added to process common color
    // dialog box information

    int result = dlg1.DoModal();
    if(result==IDOK) {
        CFourierDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        // get returned color from dialog box
        pDoc->mycolor = dlg1.GetColor();
        InvalidateRect(NULL,TRUE);
        UpdateWindow();
    }
}

```

记着，支持对话框和菜单操作的文件此处没有讨论。这些文件在 Osborne 的网站 (WWW.osborn.com) 上可以得到。

设计提示

如果读者要用这一工程的傅立叶级数段实验，不要奇怪有两个对话框，它们分别为 OnDraw() 方法的局部变量 nterms 和 mycolor 设置初值。例如 nterms 可以设置为 100，即 100 个谐波，mycolor 设置为 RGB(255,255,0)，即用黄色填充。



11.3 小结

本章使用了比前一章更复杂的 MFC Windows 程序，探索解决存在于两组学生的工程中的问题。本章学习了如何应用前面章节介绍的调试技术，以及一些检测工程中内存泄漏的强有力工具。除了使用在工程上的调试工具外，也学习了如何在调查中分析问题，学习了用有计划的方法去修复工程中的代码。

在后面两章，我们将停止讨论 MFC 类库，转向介绍 Standard Template Library(STL)。本书余下的章节再回到 MFC 类库的使用上，最后一章说明调试同时使用 MFC 和 STL 时出现的可修复的问题。 ■

第四部分

DEBUGGING
DEBUGGING

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

DEBUGGING

标准模板库(STL)

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

第 12 章 STL 编程实践

第 13 章 定位、分析和修复 STL 代码中的错误

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING



DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

第 12 章

STL 编程实践





对于 ANSI/ISO C++ STL(即 Standard Template Library, 标准模板库)是如此之新的内容, 以至许多程序员完全不知道它的存在。范围广、内容繁多的标准 C/C++ 逻辑和语法的再组合封装成了 STL, 这是 C++ 近期最重要的发展。不幸的是, 从逻辑设计到 C++ 语法和相互关系的高级使用, STL 都是如此完善, 以致于要想能够调试 STL 代码, 必须首先理解 STL。

第十二章将快速复习 C/C++ 的规范知识, 补充现今新的 C++ 关键字和语法, 然后复习类语法和模板。读完第十二章后, 对于调试 C++ 的这一最激动人心的增加部分, 就有了所需的全部概念上的理解。

12.1 多体系结构

从程序设计角度看, 今天的开发环境比十年前复杂了上百倍。它现在必须处理上百种 PC 兼容产品和其他流行的有竞争力的平台, 而不再是针对独立的 DOS 文本模式环境下的应用程序开发。

这些新的体系结构有其自身发展的操作系统和多任务、多媒体能力。另外又加上了 Internet。换句话说, 今天单个开发者使用的程序设计环境, 过去曾经是懂得多系统、通信、安全、网络和实用工具等的专家们的领域。这些专家以工作组的形式工作来保持主机的运行!

为了使程序开发者适应不断增加的资源管理的难度, 必须做一些事情。这就是, 进入 C 和 C++。这两种新的语言结合了崭新的程序设计能力, 以满足程序设计的需求。

设计提示

接近具有难以置信的能力的 C/C++ 特性的最大障碍是忽视其存在。

12.2 掌握 C++

当老板要求使用新的语言时, 绝大多数的 FORTRAN、COBOL、Pascal、PL/I 和汇编语言程序员会自学新的语言。为什么? 当然是因为公司不会给他们空余时间。他们在自己的晚上和周末努力地学习, 将其对掌握得很好的语言的理解映射到新语言的语法上。

几十年来, 程序员从老的高级语言转到下一个高级语言时, 这种方法一直在起作用。不幸的是, 到了 C 和 C++, 这种方式让刻苦、自我激励和自学的雇员发了火, 奇怪这次出了什么问题?

为了说明这一点, 此处给出一个小例子。例如, 要一个变量增 1, 在 COBOL 中, 将写成:

```
accumulator = accumulator + 1.
```


然后某天，老板对雇员说必须用 FORTRAN 写这一程序。他学习了 FORTRAN，重新写了这一语句：

```
accumulator = accumulator + 1
```

没有问题。后来公司的产品要移植到 Pascal 上，他再次自学了新的语法：

```
accumulator :=accumulator + 1
```

仍然没有问题。后来老板要求他将多年积累的程序用 C/C++ 转向 Microsoft Windows。经过废寝忘食的自学，他感觉他已经掌握了 Microsoft Windows C/C++ 的逻辑和语法，最后重写了下面的语句：

```
iaccumulator = iaccumulator + 1; //i for integer in Hungarian notation
```

然后他就被解雇了！从地方两年制学院雇佣的上级程序员看了他的代码，嘲笑他不适当的转换。确实，他学会了匈牙利命名法(C/C++ 的命名约定，就是在每个变量的名称之前加上其数据类型的缩写)，但是他只是进行了字面上的语句转换，而没有利用 C/C++ 中更有效的方法。

错误监视

将 FORTRAN、COBOL、Pascal、Basic 或 Modula-2 高级语言的语句简单地转换为 C/C++ 语句，不是真正的 C/C++ 程序设计！

他的上级程序员 Green，比他年轻 20 岁，仅知道 Microsoft Windows 的 C 和 C++ 语法，他知道该语句应该写成：

```
iaccumulator++;
```

这一语句使用了 C/C++ 增量运算符，指示编译器取消了上面的不适当“翻译”程序中的两次取码/译码，并正如变量 iaccumulator 的名字所暗示的那样，将此变量作为一个寄存器中的累加器，这是一个更高效的机器语言编码。

这一极其简单的例子只是 C/C++ 数以百计的语言特性之一，它们像流沙一样等在那里，捕捉粗心的程序员。因为 C/C++ 的这些特性，本章的重点放在与 STL 有关的必要的 C/C++ 话题上，以使读者能够完全理解和使用 STL。

本章的前半部分介绍 STL 的基本原理，后半部分复习 C/C++ 提供的那些使 STL 成为可能的关键性基础，最后以实际创建 STL 类属算法的最高级 C/C++ 语法结束。读者可能认为自己是具有足够经验的 C/C++ 程序员，那么可以跳过这一章，并且立即开始调试 STL。但可能吃惊地发现 C++ 已经发展如此之快，即使曾经系统地学过 C/C++ 课程，也可能又吃惊地发现，有很多东西教授要么不知道，要么没有讲述。而更糟糕的是——坦白地说，许多读者现在知道的 C/C++ 是不正确的。因为所有这些理由，所以笔者强烈建议读者安静地坐下来，



从头到尾读这一章。

12.3 STL——进退维谷的数据结构

在程序员正规的教育道路上，有一门通常称为“数据结构”的基础课，据统计有 50% 的逃课率。为什么？因为它涉及两个非常高深的概念，即指针和动态内存分配/释放。当这两者结合起来时，在程序开发和调试中产生的复杂性呈几何级数增长。这两个概念通常呈现为一条如此陡峭的学习曲线，以致许多程序员或者完全回避这门课，或者磕磕绊绊地混过去，然后在实际工作中再也不使用数据结构的概念。这是很令人遗憾的，因为对于程序员来说，指针和动态内存分配提供了某些最有力和最高效的算法。

下面开始学习 STL！

12.4 初识 STL

简而言之，STL 封装了 C/C++ 语言的原始能力，再加上数据结构课程中讲解的先进高效的算法，绑定成了一个简单实用的形式！在某种程度上，这类似于同计算方法课纠缠了若干年，而现在给一个高级的便携式计算器就帮助我们做了所有的工作。

可以将 STL 看作一个可扩充的框架，其中包含一些组件，如语言支持、诊断程序、通用工具、字符串、本地化、标准模板库(容器、迭代符、算法和数值量)和输入/输出。

12.5 STL 和 HP 公司

由于 C/C++ 和 Microsoft Windows 控制的环境的声望不断增长，许多第三方销售商以提供例程序库的方式发展极其有利可图的产品，这些库被设计用来存储和处理数据。为了继续维护 C/C++ 生存能力，使其成为值得选择的程序设计语言。ANSI/ISO C++ 通过严格控制 C/C++ 语言的正规定义，使其不断发展，并增加了定义这些库的一个新方法，即标准模板库 STL。

STL 是由 HP 公司的 Alexander Stepanov 和 Meng Lee 开发的，STL 被期望可以成为保存和处理数据的一个标准方法。主要编译器销售商正开始将 STL 结合到其产品中。STL 不只是作为一个次要的部分添加到世界上最流行的程序语言中，它代表一种新的革命性的能力。STL 为 C++ 程序设计语言带来一组成熟得使人惊奇的类属容器和算法，为 C/C++ 增加了新的一面。

12.6 大众化的 STL

与许多其他介绍 STL 的书简单地列举数不清的 STL 模板名字、函数、常量等不同。本章将以讲解 C/C++ 的高级语言基础开始，这使 STL 在语法上成为可能。理解了这些，即可继续学习第 13 章。

遵照这一原则，这一指导性的一章将说明允许算法类属化的语法——换句话说，C/C++ 在语法上如何将程序所做的工作与程序使用的数据类型分开，本章还将介绍类属 `void*` 指针的优缺点。总之，使用类属类型是一个比较好的方法，更好的方法是使用模板，而最好的方法是使用跨平台、可移植的标准模板 STL！

关于模板开发的一节的开头介绍了 C/C++ 用来创建对象的语法：结构。后来，结构(struct)对象的定义在逻辑和语法上发展成为 C++ 类(class)。最后，类对象变异为类属的模板。这种逐步前进的方式将使读者能够容易地吸收 C/C++ 的新特性，并为在技术上正确使用和调试 STL 代码铺平了道路。理解了这些，读者将能在逻辑和语法上理解 STL 是如何工作的，并立即开始将这项技术结合到自己的应用程序开发中。

类属的程序设计将提供如 SmallTalk 这样的语言的功能和表达方式，同时保持 C++ 的效率和兼容性。使用 STL 的程序员肯定可提高编程效率。

12.7 STL 总览

尽管 STL 的规模很大，并且其语法初看上去有些令人生畏，但实际上一旦理解其构造以及使用的元素，STL 是很容易使用的。STL 的核心是三个基础项，分别称为容器(container)、算法(algorithm)和迭代器(iterator)。这些库一起工作，可以以一种可移植的格式产生常用算法的解决方案，比如创建数组、元素插入/删除、排序和元素输出。STL 甚至还更进一步提供了内部清晰、无缝、高效的输入/输出流(iostream)集成和异常处理。

12.8 ANSI/ISO C++ 接受 STL 的过程

如果没有 ANSI C/C++ 委员会，C/C++ 编译器的多厂商实现计划早就中途夭折了。通过为 Dennis Ritchie 和 Bjarne Stroustrup 提出的 C 和 C++ 语言的正式描述补充缺少的细节，该委员会负责为我们提供可移植的 C 和 C++ 代码。直到今天，ANSI/ISO C++ 委员会继续保证 C++ 的可移植性进入新千年。

而 STL 的原创作者是 HP 公司的 Alexander Stepanov 和 Meng Lee，他们开发了支持 STL 的概念和编码。业界预期 STL 将成为保存和处理数据的标准方法。



ANSI C/C++委员会现在的标准超过了他们过去的建议，这一建议只决定为现存的实践编码和解决现存翻译程序实现中的含糊和矛盾，C/C++委员会的修改是一项革新。在大多数情况下，这些修改改善了委员会成员认为是传统 C++的不足的特性，实现了他们所赞赏的其他语言的特性和他们一直希望编程语言应有的另外一些特性。

委员会对于每一处改变都做了深入的思考和讨论，因此他们认为新的 C++定义再加上具有发展意义的 STL 定义可能是今天 C++最好的定义。

这些建议的修改大多数是由不影响现存代码的语言添加组成的。只要老的代码没有恰巧使用任何新的关键字作为标识符，那么老的程序应仍能使用新的编译器编译。然而即使在不讨论 STL 的情况下，那些有经验的 C++程序员也会对 C++有如此之多处扩展感到惊奇，例如，namespace 的使用，新型的类型转换，和运行时的类型信息等。

12.9 STL 基本组件

在概念上，STL 包含三个分开的算法问题求解工具。这三个最重要的部分是容器、算法和迭代器。容器是数据在内存中组织的方法，例如，数组、堆栈、队列、链表或二叉树。然而还有许多其他种类的容器，STL 包括那些最有用的容器。STL 容器是用模板类实现的，因此可以容易地定制它们以得到不同数据类型的容器。

所有容器有共同的管理成员函数，在其模板中定义 insert()、erase()、begin()、end()、size()、capacity()等，各容器有支持其自身需要的成员函数。

算法是应用在容器上以各种方法处理其内容的行为或功能。例如，有对容器内容排序、复制、检索和合并的算法。在 STL 中，算法是由模板函数表现的。这些函数不是容器类的成员函数。相反，它们是独立的函数。的确，STL 的令人吃惊的特点之一就是其算法如此通用。不仅可以将其用于 STL 容器，而且可以用于普通的 C++数组或任何其他应用程序指定的容器。

一组标准的算法为容器中的对象提供了检索、复制、排序、变换和数值操作功能。同一算法可用来为所有对象类型的所有容器执行某一项特别的操作。

一旦选定一种容器类型和数据行为，那么剩下唯一要做的就是用迭代器使其相互作用。可以把迭代器看作一个指向容器中元素的普通指针。可以如递增一个指针那样递增迭代器，使其依次指向容器中每一个后继的元素。迭代器是 STL 的一个关键部分，因为它将算法和容器连在一起。

12.9.1 什么是容器？

所有 STL 库的语法充分使用了 C++模板(数据类型独立的语法)。在讨论容器的类型时，要记住它们是作为模板实现的。当程序实例化容器时，其中包含的对象的类型由所给的模板

参数决定。有三种主要的容器类型，向量(即动态数组)、双端队列(deque)和线性列表，还有专用的容器类型：位集(bitset)、映射和多映射(multimap)。

序列容器以线性方式存储有限组同一类型的对象。一个名字数组就是一个序列容器。要为某一特定应用程序使用哪种序列类型容器——向量、列表还是双端队列，取决于其检索要求。

12.9.1.1 向量类

向量序列允许随机存取数据。向量带有实例计数器或指针以指示向量序列的结尾，其元素是邻接的同类对象。通过使用下标操作，使随机存取容易执行。向量序列允许在动态结构的结尾添加或删除项目，而不会有过多的开销。然而在中间插入和删除自然要多消耗一些时间，因为移动其他项目以给新加的或删除的项目让出位置要花时间。

12.9.1.2 列表类

列表序列提供双向存取，它允许在任何处执行插入和删除而没有过多的系统开销。随机存取是通过向前或向后迭代至目标对象模拟的，列表是由向前和向后指针链接非邻接对象组成的。

12.9.1.3 双端队列类

双端队列序列与向量序列类似，只是双端队列序列允许在容器的开头和结尾快速地执行插入和删除操作，随机插入和删除的效率要低一些。

12.9.1.4 位集类

位集类支持一组位操作，如 flip()、reset()、set()、size()和 to_string 等。

12.9.1.5 映射类

映射类提供了联合容器，它具有唯一的關鍵字，映射到特定的值。

12.9.1.6 多映射类

多映射类在原始的功能上类似于映射类，除了一个很小的差别：没有唯一的關鍵字映射特定的值。

12.9.2 什么是适配器？

STL 支持三种适配器容器，可以将其和前面列出的三种序列容器结合在一起。方法是这样的：首先，选择一个适当的程序指定的容器；接着通过在声明中命名这一已存在的容器



来实例化一个容器适配器类:

```
queue< list< bank_customer_struct > >TellerOneQueue;
```

这一例子通过使用列表容器作为底层数据结构(这是围绕一个假定的等待出纳员的银行顾客建立的),实例化了一个队列容器,它是 STL 支持的三种适配器容器之一。

容器适配器隐藏了底层容器的公共接口并实现其自己的接口。例如,队列数据结构类似于列表,但要求有自己的用户接口。STL 合并了三种标准适配器容器:堆栈、队列和优先级队列(priority_queue)。

12.9.2.1 堆栈类

堆栈适配器提供了逻辑操作 push()和 pop(),使标准的后进先出,即 LIFO 解决方案成为可能。堆栈的了不起之处在于给出了某些问题的解决方案,例如,求一个中缀(Infix)算术表达式的值,为了明确求值,该表达式被转换为后缀(Postfix)形式。

12.9.2.2 队列类

无论存储序列容器是向量还是链表,队列适配器采用这种底层方案,用方法 push()在列表的结尾添加项目,用方法 pop()在列表的开头删除项目。队列算法的首字母缩写词为 FIFO,即先进先出。

12.9.2.3 优先级队列类

优先级队列与队列适配器类似,所有添加进队列的项目都在列表的结尾。然而,不像队列适配器那样只从列表的头部删除项目,优先级队列适配器首先删除列表中优先级最高的项目。

12.9.3 什么是算法?

与容器适配器类似,算法也作用于容器。算法为容器提供了实例化、排序、检索和数据变换功能。有趣的是,算法不是作为类的方法实现的,而是作为独立的模板函数实现的。因此,算法不仅作用于 STL 容器,而且作用于标准 C++数组或程序员自己创建的容器类。

典型的算法包括: find(),定位一个特定的项目; count(),返回列表中项目的个数; equal(),用来进行比较; search(); copy(); swap(); fill(); 以及 sort()。

12.9.4 什么是迭代器?

无论何时,应用程序需要在容器的元素中移动,都要使用迭代器。迭代器和用来访问个别数据项的指针类似。在 STL 中,迭代器用一个迭代器类的对象表示。可以用 C/C++增

量运算符“++”递增一个迭代器，将其移入下一个元素的地址。也可以使用取内容运算符“*”访问当前项目中的个别成员，专门的迭代器能够记住指定容器元素的位置。

有几种不同的迭代器类，它们必须与特定的容器类型一起使用。主要的三个迭代器类是：向前、双向和随机存取迭代器。

- 向前迭代器在容器中只能向前进，每次一个项目。向前迭代器不能向后移动，也不能被更新以指向容器中间的任何位置。
- 向后迭代器的工作方式与对应的向前迭代器类似，只是向后移动。
- 双向迭代器可以向前或向后移动，而不能被赋值或更新以指向容器中间的任何元素。
- 随机存取迭代器比双向迭代器更进一步，允许应用程序在容器内执行任意的位置跳转。

另外，STL 定义了两个专用迭代器，即输入和输出迭代器。输入和输出迭代器可以指向特定的设备；例如，一个输入迭代器可以指向用户指定的一个输入文件，即 `cin`，然后将其顺序读入容器。同样，一个输出迭代器可以指向用户指定的一个输出文件即 `cout`，并执行相反的逻辑操作——顺序输出容器的元素。

不像向前、向后、双向和随机存取迭代器，输入和输出迭代器不能保存其当前地址。向前、向后、双向和随机存取迭代器中必须存放有元素地址，以便于知道其在容器中的位置。而由于输入和输出迭代器是指向设备的指针，在结构上不表现同样类型的信息，因此没有地址记忆功能。

12.9.5 其他的 STL 组件

除了容器、算法和迭代器之外，STL 还定义了另外几种组件：

- 分配算符，为单个容器管理内存分配。
- 谓词，它们在本质上是一元的或二元的，意即其作用于一个或两个操作数并总是返回“真”或“假”。
- 比较函数，一种独特的二元谓词，比较两个元素并只在第一个参数小于第二个时返回“真”。
- 函数对象，包括加、减、乘、除、取模、取反、等于、不等于、大于、大于或等于、小于、小于或等于、逻辑与、逻辑或以及逻辑非。

12.10 完整的 STL 程序包

现在将 STL 的结构组件，在逻辑上归纳为以下几类：

A. STL 头文件可以分成三个主要的组织概念：

1. 容器是支持普通数据组织方法的模板类：



<deque>、<list>、<map>、<multimap>、<queue>、<set>、<stack>和<vector>。

2. 算法是对对象序列执行普通操作的模板函数，包括：<algorithm>、<functional>和<numeric>。

3. 迭代器是把算法和容器粘在一起的黏合剂，包括<iterator>、<memory>和<utility>。

B. 输入输出包括以下各项内容的组件：

1. 输入输出流的向前声明<iosfwd>。
2. 预定义输入输出流对象<iostream>。
3. 基输入输出流类<ios>。
4. 流缓冲<streambuf>。
5. 流格式和操纵器：<iosmanip>、<istream>和<ostream>。
6. 字符串流<sstream>。
7. 文件流<fstream>。

C. 其余标准 C++ 头文件包括：

1. 语言支持包括：
 - a. 在整个<cstddef>库中使用的普通类型定义的组件。
 - b. 预定义类型<limits>、<cfloat>和<climits>的特征。
 - c. 支持 C++ 程序开始和结束的函数<cstdlib>。
 - d. 对动态内存管理的支持<new>。
 - e. 对动态类型标识符的支持<typeinfo>。
 - f. 对异常处理的支持<exception>。
 - g. 其他运行时的支持<cstdarg>、<ctime>、<csetjmp>和<csignal>。
2. 诊断包括以下各项内容的组件：
 - a. 报告几种异常情况<stdexcept>。
 - b. 用文件证明程序断言<cassert>。
 - c. 错误数字代码的全局变量<cerrno>。
3. 字符串包括以下各项内容的组件：
 - a. 字符串类<string>。
 - b. 以 NULL 结尾的顺序工具：<cctype>、<cwctype>和<wchar>。
4. 文化语言组件包括：
 - a. 对字符分类和字符串校对，数字、货币和日期/时间的格式和分析，以及消息检索的国际化支持，使用<locale>和<locale>。

12.11 杂乱的 C/C++ 家族

C/C++ 编程领域真是乱糟糟的，有“早期 C”、ANSI C、C++、ANSI C++、ANSI/ISO C++、Borland International 公司的 C/C++、Microsoft 的 C/C++，甚至还可能有某个自修大学的教授创造的 C/C++ 版本，更别提以后还会有什么稀奇古怪的版本。

学习 C/C++ 的最大问题是找到一个声誉好的版本。除了极少数例外情况，许多程序员是自学 C/C++ 的。他们是专业级程序员，具有多年的经验，从学院教的语言自学到今天流行的任何语言。因此他们的方法通常是从 COBOL，到 FORTRAN，到 PL/I，到 Pascal，到 Modula-2，连续不断地学下去。

这一方法在过去实际上有很好的效果，因为所有老的高级语言有基本相同的特性，只是语法上的区别。现在把这一自学成才的程序员推进 GUI(图形用户界面)环境中，再加上要使用 C/C++ 和面向对象技术开发多任务程序，结果只能是一团糟。

C 和 C++ 提供了如此多的新语言特性、设计思想和复杂的语法，以致于将其他语言的理解映射过来是行不通的。

好消息是，给一点指导性提示，读者即可凭目前所知道的任何编程语言快速掌握这门最新的技术。这就是本章的内容。现在开始吧！

12.12 回顾数据结构

包含在 STL 中的代码是极其高效的。此处“高效”的意思是指使用动态分配内存创建对象，和使用指针跟踪。本节复习用 C/C++ 建立块，使上述成为可能。

12.12.1 静态与动态

首先，要理解本节使用的单词 `static` 不是 C/C++ 关键字 `static`，这很重要。相反，`static` 用来描述某种内存分配。由于大多数程序员更容易理解代码，所以下面给出一个例子：

```
void main( void )
{
    int ivalue;    //i(nt) value
    .
    .
    .
}
```

这段代码声明了一个整型变量 `ivalue`。现在考虑一下，`ivalue` 的存储单元是在加载程序



时分配的，并且这一内存分配一直持续到程序退出。这是一个静态内存分配的例子。

静态内存分配不受运行时程序的控制，而是受加载时的控制。程序员不能得到更多 *ivalue*，也不能在运行时删除 *ivalue* 的内存分配。这是这种存储类型的不足之处。

动态内存分配具有受运行时控制的优点。令人遗憾的是，其语法不是那么直接，需要使用指针(注意：下面的例子只是为了强调静态和动态内存分配的差别，并不需要作为一个实际的例子)。

```
void main( void )
{
    int *pivalue;           // P(ointer to)I(nt)value;
    int iLoopControl, iAsManyAsUserWants;
    cout << "How many integers would you like to create at run-time?";
    cin >> iAsManyAsUserWants;
    for( iLoopControl = 0, iLoopControl < iAsManyAsUserWants, iLoopControl++;
        pivalue = new int; // pivalue set to address of
        // run-time dynamically allocated RAM
    )
    {
        // ...
    }
}
```

在这一例子中，变量 *pivalue* 不是一个整型变量，而是一个指针变量，可以保存一个内存单元地址，这一存储单元足够大到存储一个整型变量。实际整型大小的内存分配是通过 C++ 关键字 **new** 完成的。C 程序员可以认为 **new** 等同于 **malloc()** 或 **calloc()**。

最重要的是，注意最终用户在运行时，可以根据自己的需要选择整型数的个数。用户也可以在运行时，通过简单地使用 C++ 关键字 **delete** 选择要删除多少个整型数，比如：

```
delete pivalue;
```

可以清楚地看出在运行时对内存分配/释放的控制有极高的效率，应用程序决不会超出其当前需要而抢夺系统内存资源，STL 充分地利用了这一点。

12.12.2 类型指针

和 **int**、**float** 和 **char** 类型的普通变量不同，指针变量不存放数据本身，它们保存数据的地址。在上一部分，可以看到与动态内存分配结合时，这一概念多么有效。然而这两个概念产生了一个问题：类型检验！

在设计上，C 和 C++ 不是强类型语言。这意味着两个编译器都将接受如下语句：

```
char cvalue = 65;    // initializing an integer variable with a integer
和：
```

```
int ivalue = 'A';    // initializing an integer variable with a character
```

但是，当遇到指针时，C 和 C++ 都变成强类型语言。例如，浮点类型的动态分配内存单元地址不能赋给整型类型的指针，比如：

```
int *pivalue;           // p(ointer to) i(nt) value;
pivalue = new float;    // illegal attempt to
                        // assign a float address to int pointer
```

从一开始, C 和 C++ 就围绕类型检验有一个依照语法的方式, 即使用标准类型 `void`。

12.12.3 void 指针

当 C/C++ `void` 数据类型与指针变量定义结合在一起时, 指示编译器定义的变量保存的不是数据本身, 而是一个地址。但是, 它没有指示编译器指向的是什么数据类型! 这可以产生强有力的编程方法。

在下面的例子中, 一个子程序用于输出三个动态分配的数据类型之一:

```
#include <iostream>
using namespace std;
void printit ( void *pData, char cRunTimeChoice );
void main ( void )
{
    char *pchar, cRunTimeChoice;
    int *pivalue;
    float *pfvalue;
    cout << "Please enter the dynamic data type you want to create\n"
         << " press c for char, i for int, or f for float: ";
    cin >> cRunTimeChoice;
    switch ( cRunTimeChoice ) {
        case 'c': pchar = new char;
        cout << "\nEnter a character: ";
        cin >> *pchar;
        printit ( pchar, cRunTimeChoice );
        break;
        case 'i': pivalue = new int;
        cout << "\nEnter an integer: ";
        cin >> *pivalue;
        printit ( pivalue, cRunTimeChoice );
        break;
        default: pfvalue = new float;
        cout << "\nEnter a float: ";
        cin >> *pfvalue;
        printit ( pfvalue, cRunTimeChoice );
    }
}

void printit ( void *pData, char cRunTimeChoice )
{
```



```
cout << "\nThe Dynamic Data type entered was ";
switch ( cRunTimeChoice ) {
    case 'c': cout << "char and a value of: "
                  << *(char *)pData;
                break;
    case 'i': cout << "int and a value of: "
                  << *(int *)pData;
                break;
    default: cout << "float and a value of: "
                  << *(float *)pData;
}
delete pData;
}
```

理解 void 指针的关键语句是 *printit()* 原型:

```
void printit ( void *pData, char cRunTimeChoice );
```

对 *printit()* 的三次调用:

```
printit ( pchar, cRunTimeChoice );
printit ( pival, cRunTimeChoice );
printit ( pfval, cRunTimeChoice );
```

注意我们是如何正式告诉编译器 *printit()* 的第一个形式参数类型是 void 指针(void*)的。在函数的形参列表和调用语句的实参之间暂停了正常的类型检验——这是程序能够执行的唯一原因。

然而与 int*或 float*指针变量声明不同, 使用 void*的语法不指定指针变量指向的数据类型。例如:

```
void *pToWhoKnowsWhat;
```

这有好处也有坏处。好处是当将实际地址赋给它时, 编译器不执行任何类型检验; 坏处也是编译器不执行任何类型检验。更糟糕的是, 程序不能用 void*类型指向任何变量。这解释了 *printit()* 的 switch-case 语句中的三个类型转换语句为什么将一个 void 指针转换为特定的指针类型。

```
*(char *)pData;
*(int *)pData;
*(float *)pData;
```

没有类型检验, 如果代码意外地赋了错误的地址类型, 则代码编译:

```
void * pival; // variable name indicates it will hold an int type address
...
```

```
pivalue = new float; // incorrect assignment of RAM float precision!
```

前面的代码部分声明 *pivalue* 为一个 **void*** 指针，然而变量的名字隐含它将保存一个整型变量的地址(*p(ointer to)i(nt)*)，然后该指针实例化为一个浮点型内存单元的地址。这是完全合法的代码，对于某些程序甚至在逻辑上都很好，但是查看下面这两条语句：

```
some_function( pivalue );    // call to some function
...
void some_function( int * pivalue ); // function prototype
```

首先，这一代码可能编译也可能不编译，这取决于编译器的错误/警告等级设置，因为编译器(C/C++)认出 *pivalue* 的正式声明 **void*** 与函数的第一个形参类型 **int*** 不匹配。记住，指针的类型是缺省的，编译器在编译时识别这种不匹配。但是一个聪明的 C/C++ 程序员可能把调用语句重新写为：

```
some_function( (int *)pivalue); // working call statement!
...
void some_function(int * pivalue);
{
    // sample function body code...
    cout << "The integer value is: " << *pivalue; // outputs garbage
```

执行这种程序将引起 *some_function()* 输出一堆垃圾，因为内存单元包含一个 IEEE 浮点编码值，而函数指示编译器将其作为一个四字节整型值译码。

错误监视

永远记住，编译器为 **void*** 类型的函数形参编译时不执行类型检验，这可能导致运行时的大错！

然而，这段代码编译通过了，没有警告也没有错误，这实在是个坏消息。这种 **void** 指针可能产生调试时和运行时的错误。解决的方法是 C++ 模板。

12.13 复习匈牙利命名法

测试的问题：下面的代码段有错误否？

```
int operandA = 1, operandB = 2;
float result;
...
result = operandA / operandB
```

如果回答有错，那就答对了。但是，如果认为语句是正确的，那好，我们说幸好有 Charles



Simony 发明的匈牙利命名法。

初始的问题是由 C/C++ 的除法运算符 “/” 引起的，它是各种数据类型的重载。在许多其他程序语言中，整型除法和浮点型除法使用两个分开的运算符。例如，Pascal 使用 “/” 作为浮点精度的运算符，用 “div” 作为得到整型结果的运算符。显然，一个 Pascal 程序员可以容易看清表达式使用了哪个运算符。不幸的是，对于粗心的 C/C++ 程序员没有类似直观的线索。而 C/C++ 除法运算符自己检查每个操作数的数据类型，然后决定是执行整型除整型得到整型结果，还是浮点型除浮点型得到浮点型结果。上面的表达式为变量 *result* 赋的值为 0，而不是逻辑上需要的 0.5 (由变量 *result* 的类型是浮点型可以看出)，这是因为 *operandA* 和 *operandB* 的数据类型是整型。

现在想象，将这一表达式嵌入一个大程序中，它错误地计算了值。而我们不是代码的作者，但必须捕捉这一问题。下面介绍匈牙利命名法，在匈牙利命名法中，每个标识符的名字 (变量、常量等的名字) 以其数据类型的缩写开头。此处是一些例子 (注意指针变量的排列 *piValue*、*pfValue* 和 *pszLastName*，把指针操作符 “*” 放在其他数据类型的前边，强调这些变量是指向数据的指针，而不是数据本身)：

```
char      cMenuSelection;
int       iValue;
float     fValue;
char      szLastName[ MAX_LETTERS + NULL_STRING_TERMINATOR ];
int       *piValue;
float     *pfValue;
char      *pszLastName = szLastName;
double    dValue, *pdValue = &pdValue;
long double ldValue, *pldValue = &pldValue;
```

现在设想将原代码段重新编写为：

```
int iOperandA = 1, iOperandB = 2;
float fResult;
...
fResult = iOperandA / iOperandB
```

在这种形式中，一个有经验的 C/C++ 程序员看到除法运算符作用于两个整型变量，而结果精度为浮点型，将立即怀疑表达式可能产生数字计算错误。

对于在 Microsoft Windows 对象中或 IBM OS/2 对象中的成百上千行代码，匈牙利命名法将大大帮助程序员密切理解算法中的数据类型。这不论在领会程序的逻辑上，还是在标记调试断点上，都将大大节省时间。

设计提示

指向数据的指针比静态变量更高效，void 指针聪明地解决了编译时形式参数对实际参

数的阻碍，void 指针存在的问题最终要靠模板化的指针解决。

12.14 函数重载

C++的许多面向对象的概念有程序上的基础。函数重载是一个程序上的概念，它允许程序员使用同一名字定义几个函数。这一语法只要求每个函数的形参表是唯一的，唯一是指参数的个数、参数的顺序和参数的数据类型。看下面几个函数原型的例子：

```
int averageArray(int iarray[]);
float averageArray(float farray[]);
double averageArray(double darray[]);
```

除了语法简明外，适当的函数重载还符合逻辑上的需要。通常，这包含函数体算法的重复，它们完成同样的功能，但经常对于不同的数据类型——对于本例，求数组元素的平均值。第一个子程序处理整型数组元素，第二个处理浮点型数组元素，第三个处理双精度型数组元素。然而，三个函数都是求数组元素的平均值！

24x7

重载函数的返回值类型在重载定义中的“唯一”中不起任何作用。因此，下面的重载函数原型是非法的：

```
void averageArray(int iarray[]);
int averageArray(int iarray[]);
float averageArray(int iarray[]);
```

函数重载是程序设计上一个强有力的工具，它 also 支持重载类成员函数(本章后面将回顾类)。

12.15 函数指针

迄今为止所有的例子说明了各种数据可以如何被指针引用。事实上，也可以通过使用指向函数的指针访问代码部分。指向函数的指针与指向数据的指针有同样的用途，也就是说，允许函数被间接引用，正如指向数据的指针允许数据被间接引用一样。

指向函数的指针可以有许多重要的用处。例如，考虑函数 `qsort()`。函数 `qsort()` 需要一个指向函数的指针作为其参数。被引用的函数中包含数组元素排序时执行的必要的比较。编写 `qsort()` 时就需要一个函数指针，因为函数间的比较过程可能是一个复杂的过程，超过了一个单独的控制标志的范围。不可能以值传递一个函数，然而 C/C++ 支持传递指向数据的指针或指向函数的指针。



函数指针的概念通常使用编译器提供的 `qsort()` 函数为例子说明。不幸的是，在许多情况下，函数指针被声明为指向其他内部函数。下面的 C 和 C++ 程序示范如何定义一个指向函数的指针以及如何转变程序员自己的函数以使其能被传递给标准库(`stdlib`)函数 `qsort()`。此处是该 C++ 程序：

```
// qsort.cpp
// A C program illustrating how to declare your own
// function and function pointer to be used with qsort( )
// Chris H. Pappas and William H. Murray, 2000
//
#include <iostream>
#include <stdlib>
#define IMAXVALUES 10
int icompare_func(const void *iresult_a, const void *iresult_b);
int (*ifunct_ptr)(const void *,const void *);
void main( )
{
    int i;
    int iarray[IMAXVALUES]={0,5,3,2,8,7,9,1,4,6};
    ifunct_ptr=icompare_func;
    qsort(iarray,IMAXVALUES,sizeof(int),ifunct_ptr);
    for(i = 0; i < IMAXVALUES; i++)
        cout <<[["|"]] << iarray[i];
}
int icompare_func(const void *iresult_a, const void *iresult_b)
{
    return((* (int *)iresult_a) - (*(int *)iresult_b));
}
```

函数 `icompare_func()` (将被称为引用函数) 的原型符合函数 `qsort()` (将被称为调用函数) 的第四个参数的要求。

也就是说，函数 `qsort()` 的第四个参数必须是一个函数指针，被指向的函数(即引用函数)必须被传给两个常量 `void*` 类型的参数(注意：记住关键字 `const` 在形参列表中的位置锁定的是指向的数据，而不是指向数据的地址。这意味着即使比较程序能够执行而不能正确排序，它也不能破坏数组的内容！)这是因为 `qsort()` 为排序的比较算法使用了引用函数。现在我们理解了引用函数 `icompare_func()` 的原型，下面再花一些时间研究这一引用函数的函数体。

如果引用函数的返回值小于零，那意味着引用函数的第一个参数小于第二个参数。返回零值表明参数值相等，返回值大于零表明第一个参数比第二个大。所有这些是由 `icompare_func()` 中唯一的一条语句完成的：

```
return((* (int *)iresult_a) - (*(int *)iresult_b));
```


因为两个指针都是作为 `void*` 类型传递的，所以它们被转换为正确的指针类型 `int*`，然后执行取内容操作(*)。指向的两个值相减的结果的类型符合 `qsort()` 的比较准则。

以上讨论了函数 `qsort()` 对 `icompare_func()` 的函数原型的要求，这实际上是 `icompare_func()` 函数原型下面的指针函数声明：

```
int icompare_func(const void *iresult_a, const void *iresult_b);
int (*func_ptr)(const void *, const void *);
```

一个函数的类型是由其返回值类型和参数列表特征决定的，指向 `icompare_func()` 的指针必须确定指向的函数的类型，你可能因此想到下面的语句将实现这一点：

```
int *ifunc_ptr(const void *, const void *);
```

这条语句几乎是正确的。问题是编译器将这一语句解释为函数 `ifunc_ptr()` 的定义，该函数有两个参数并返回一个 `int*` 类型的指针。也就是说，与运算符“*”结合的是类型指定符 `int`，而不是 `ifunc_ptr()`。必须用圆括号将运算符“*”和 `ifunc_ptr()` 结合在一起。

正确的语句将 `ifunc_ptr()` 声明为一个指向函数的指针，该函数有两个参数和一个整型返回值。这样 `ifunc_ptr()` 的类型就符合 `qsort()` 的第四个参数的要求了。

现在 `main()` 函数体中，剩下唯一要做的就是将 `ifunc_ptr()` 初始化为函数 `icompare_func()` 的地址。`qsort()` 的参数是被排序的表的基元素，即第零个元素的地址(`iarray`)，表中项的个数(`IMAXVALUES`)，每个表元素的大小(`sizeof(int)`)，和指向比较函数的指针(`ifunc_ptr()`)。

学习理解函数指针的语法可能有一些挑战性。下面看一些例子。此处是第一个：

```
int *(*ifunc_ptr)(int)[5];
float *(*ffunc_ptr)(int,int)(float);
typedef double *(*dfunc_ptr)() [5]();
dfunc_ptr A_dfunc_ptr;
(*(*function_ary_ptrs)() [5])();
```

第一条语句将 `ifunc_ptr()` 定义为一个指向函数的指针，该函数被传递一个整型参数并返回一个指向数组的指针，数组的元素为五个整型指针。

第二条语句将 `ffunc_ptr()` 定义为一个指向函数的指针，该函数带两个整型参数并返回一个指向函数的指针，函数有一个浮点型参数并返回浮点型值。

使用 `typedef` 声明语句，可以避免复杂声明的不必要重复。上面的 `typedef` 声明是这样的：`dfunc_ptr()` 被定义为一个指向函数的指针，该函数没有参数而返回一个指向数组的指针。数组的元素为五个整型指针，这五个指针都指向函数，这些函数没有参数并返回双精度型值。

最后的语句是一个函数声明，而不是变量声明。这条语句将 `function_ary_ptrs()` 定义为一个函数，该函数没有参数并返回一个指向数组的指针。数组的元素为五个指向函数的指针，这些函数没有参数并返回整型值。外层的函数的返回类型(整型)是 C/C++ 缺省的。



在实际工作中，我们将很少遇到这样复杂的声明和定义。然而，如果理解这些声明，那么在分析平常的变化种类时就很有信心了。

12.16 运算符重载

在本章的前一部分，学了重载普通函数和类成员函数的方法。在这一节，将学习重载 C++ 的运算符。在 C++ 中，能够为类中类似于 +、-、*、/ 这样的运算符赋以新的定义。

运算符重载的思想在众多程序设计语言中是普遍的，即使没有明确地实现。例如，所有的编译语言都可以使用 “+” 运算符对两个整数、两个浮点数或两个双精度数(或其等价的类型)执行加法运算。这就是运算符重载的本质——对于不同数据类型使用同样的运算符。在 C++ 中可以进一步扩展这一概念。例如，在大多数编译语言中，使用 “+” 运算符执行复数、矩阵或字符串相加是不可能的。

下列操作在所有的程序设计语言中是合法的：

```
3+8
3.3+7.2
```

下列操作通常是不合法的操作：

```
(4-j4)+(5+j10)
(15°20'15")+(53°57'40")
"combine"+"strings"
```

如果后 3 个操作使用 “+” 运算符是可行的，那么当设计新程序时，程序员的工作量将大大减少。可喜的是，在 C++ 中，“+” 运算符可以重载，上面三种操作能够成为合法的，许多其他的运算符也可以被重载。在 C++ 中，广泛地使用运算符重载，可以在各种 Microsoft C++ 库中找到例子。

12.16.1 运算符和函数调用的重载

在 C++ 中下列运算符可以被重载：

+	-	*	/	=	<	>	+=	-=
*=	/=	<<	>>	>>=	<<=	==	!=	<=
>=	++	--	%	&	^^	!	!	~
&=	^=	=	&&		%=	[]	()	new
delete								

主要的限制是运算符的语法和优先级与其原来的含义必须保持一致，另外重要的一点是运算符重载只能在类的范围内完成。

12.16.2 编写自己的重载运算符

重载一个运算符时，使用关键字 `operator`，其后面紧跟运算符本身：

```
type operator opr(param list)
```

例如：

```
angle_value operator + (angle_argument);
```

此处，`angle_value` 是类的名字，后面是关键字 `operator`，然后是运算符“+”，最后是传递给重载运算符的参数。

在类 `angle_value` 内，可以直接相加几个以度/分/秒形式定义的角度：

```
angle_value angle1("37° 15' 56\"");
angle_value angle2("10° 44' 44\"");
angle_value angle3("75° 17' 59\"");
angle_value angle4("130° 32' 54\"");
angle_value sum_of_angle;
sum_of_angle=angle1+angle2+angle3+angle4;
```

如同从前面一个例子知道的，秒的符号是双引号(")，这个符号也用于标志字符串的开始和结束。如果在引号前加一个反斜杠，引号即可显示在屏幕上。本书使用这种格式输入数据。

在这个程序中，还必须重视另一个问题：从秒到分和从分到度的进位信息必须适当地处理。当秒数或分数超过 59 时，发生进位。当然这与运算符重载没有任何关系。下面是程序中处理的方法：

```
// opover.cpp
// C++ program illustrates operator overloading.
// Program will overload the "+" operator so that
// several angles, in the format degrees minutes seconds,
// can be added directly.
// Chris H. Pappas and William H. Murray, 2000
//
#include <strstream>
#include <stdlib>
#include <string>
using namespace std;
class angle_value {
    int degrees, minutes, seconds;
public:
    angle_value() {degrees=0,
```



```
        minutes=0,
        seconds=0;} // constructor
angle_value(char *);
angle_value operator +(angle_value);
char * info_display(void);
};
angle_value::angle_value(char *angle_sum)
{
    degrees=atoi(strtok(angle_sum,"°"));
    minutes=atoi(strtok(0,"' "));
    seconds=atoi(strtok(0,"\\\""));
}
angle_value angle_value::operator+(angle_value angle_sum)
{
    angle_value ang;
    ang.seconds=(seconds+angle_sum.seconds)%60;
    ang.minutes=((seconds+angle_sum.seconds)/60+
        minutes+angle_sum.minutes)%60;
    ang.degrees=((seconds+angle_sum.seconds)/60+
        minutes+angle_sum.minutes)/60;
    ang.degrees+=degrees+angle_sum.degrees;
    return ang;
}
char * angle_value::info_display()
{
    char *ang[15];
    // stringstream required for incore formatting
    ostrstream(*ang,sizeof(ang)) << degrees << "°"
        << minutes << "'"
        << seconds << "\\\""
        << ends;

    return *ang;
}
main()
{
    angle_value angle1("37° 15' 56\\\""); //make with alt-248
    angle_value angle2("10° 44' 44\\\"");
    angle_value angle3("75° 17' 59\\\"");
    angle_value angle4("130° 32' 54\\\"");
    angle_value sum_of_angles;
    sum_of_angles=angle1+angle2+angle3+angle4;
    cout << "the sum of the angles is "
        << sum_of_angles.info_display() << endl;
```

```
return (0);
}
```

度分秒混合相加的详细情况包含在声明“+”运算符重载的一小段代码中:

```
angle_value angle_value::operator+(angle_value angle_sum)
{
    angle_value ang;
    ang.seconds=(seconds+angle_sum.seconds)%60;
    ang.minutes=((seconds+angle_sum.seconds)/60+
                minutes+angle_sum.minutes)%60;
    ang.degrees=((seconds+angle_sum.seconds)/60+
                minutes+angle_sum.minutes)/60;
    ang.degrees+=degrees+angle_sum.degrees;
    return ang;
}
```

此处用除法和取模运算完成求和, 以保证正确的进位。

程序操作的更详细情况被忽略了, 因为在更早的例子中已经看到了大部分函数和模块。但是要记住, 当重载运算符时, 必须保证正确的运算符语法和优先级。

这一程序的输出显示了四个角度的求和值如下:

```
the sum of the angles is 253° 51' 33"
```

这一答案正确否?

24x7

掌握 C 和 C++所提供的一些关键性基础内容可以使我们理解 STL 是如何施展其魔力的!

12.17 从结构到模板

STL 出色地利用了指针、函数重载和运算符重载, 另外还十分依赖于模板。在查看一个 STL 例子之前, 先查看下面这一简明易懂的 C 代码段, 它使用了 `#define` 和连接符(`##`)预处理语句定义了一个二叉树节点:

```
C Example
#define BINARY_TREE( t )
typedef struct _tree_##t {
    t data;
    struct _tree_##t *left;
    struct _tree_##t *right;
} BINARY_TREE_##;
```



注意预处理器是如何用用户选择的数据类型替换参数 `t` 的，例如：

```
BINARY_TREE( int );  
BINARY_TREE( float );  
BINARY_TREE( my_structure );
```

然后，程序完全重定义了这一节点。例如，对于 `int` 数据类型，二叉树节点的定义将变为：

```
typedef struct_tree_int {  
    int data;  
    struct _tree_int *left;  
    struct _tree_int *right;  
} BINARY_TREE_int;
```

这只是个有关 C/C++ 语言提供的固有模块性的小例子。上面的几个例子在 C 中都是合法的，不需要附加 C++ 的任何语法！

但是，这一聪明的例子存在一个内在的问题：与可以用于产生宏的内联(**inline**)函数不同，**#define** 定义的宏没有错误检查能力。**#define** 语句所做的是由 C/C++ 的两次编译之一严格地完成字符串查找和替换操作。显然，为了产生可靠和可移植的代码，必须有一些其他的方法，以产生可靠的定义，这一方法就是 C++ 模板。

12.17.1 template 关键字

模板是在 ANSI/ISO C++ 标准化处理以前添加到 C++ 中的最后的特性之一。正如 Bjarne Stroustrup(C++ 的作者之一)说的那样，“模板被认为是设计适当的容器类所必需的。对许多人来说，C++ 最大的一个问题是缺少可扩展的标准库。为产生这样的库，最大的问题是 C++ 没有提供一个充分通用的工具用于定义诸如列表、向量和联合数组这样的“容器类”。将模板结合到 C++ 语言中直接导致了 STL 的出现，STL 是使用模板类和模板函数的容器类和算法的标准化库。

12.17.2 模板语法

作为一个程序员，一定理解函数和函数调用的概念。函数包含一个模块化设计、可重复使用、求解单一问题的算法。在调用程序的算法执行中，函数调用为正在执行的程序传递需要的实际参数。

C++ 模板以完全新的方式使用参数：创建新函数和类。与为函数传递参数不同，模板是在编译时创建这些新函数和类的，而不是在运行时。

模板的语法为：

```
template <argument_list> declaration
```

在关键字 `template` 和参数列表(argument_list)之后, 程序员提供模板声明。在此处, 定义参数化的类或函数的版本。在使用模板时, 由 C++编译器负责基于传递给模板的参数产生类或函数的不同版本。

12.17.3 模板函数

要理解模板如何在 STL 中发挥作用, 需要知道有两种类型的模板: 类模板和函数模板。函数模板产生函数, 而类模板产生类。

下面的例子定义了一个求任何类型的数据平方的函数模板:

```
template <class Type>
Type squareIt( Type x ) {return x*x;} //function template
```

可以为函数模板 `squareIt()` 传递任何合适的数据类型: 例如:

```
void main( void )
{
    cout << "The square of the integer 9 is: " << squareIt( 9 );
    cout << "The square of the unsigned int 255 is: " << squareIt( 255U );
    cout << "The square of the float 10.0: " << squareIt( 10.0 );
    //...
}
```

这三条语句使编译器在编译时产生如下三个独立的函数体, 一个是整型数据的实例, 另一个是无符号整型的, 第三个是浮点型的:

```
int square(int x) { return x * x; }
unsigned int square(unsigned int x) {return x * x; }
double square(double x) { return x * x; }
```

12.17.4 模板类

第二类模板是类模板, 下面的例子定义了一个简单的数组容器类模板:

```
template < class Type, int MAX_ELEMENTS >
class Array {
protected:
    Type *pTypeArray;
public:
    Array() { pTypeArray = new Type[ MAX_ELEMENTS ]; } //constructor
    ~Array() { delete[] pTypeArray; } //destructor
    //...
};
```



这一模板类定义创建了一个任意类型的数组容器类！模板的第一个参数定义数组存放的元素的类型，第二个参数定义数组保存元素的个数。这一数组的类型可以是任意的，从标准 C++ 的简单数据类型，比如 `int`，到应用程序指定的复杂的结构或复杂对象。

程序实例化一个实际模板类定义的实例，使用几乎与函数一样的语法：

```
void main( void )
{
    Array < float, 10 > fArray;
    Array < int, 25 > iArray;
    Array < MY_STRUCTURE_DEFINITION, MAX_RECORDS > strucArray;
    Array < MY_CLASS_DEFINITION, iRunTimeUsersChoice > classArray;
    //...
}
```

对于每个实例，编译器为传递给数组类模板的每一种数据类型都产生一个崭新的类版本。无论实例出现在何处，编译器都在编译时完成参数替换。

12.18 为什么 STL 比模板好

理论上，C++ 模板满足了对容易使用的容器类的需要。但在实际中，它并不总是那么简单，因为另外尚有几个问题。首先，由于模板容器类的实现中的固有问题，基于模板的容器可能明显地比对应的基于 C 的慢。例如，许多基于模板的容器类依赖于继承性，而某些继承性可显著地减慢程序。

另一个问题是兼容性。如果碰巧使用来自不同销售商的两个模板，可能在其之间有兼容性冲突，因为模板没有标准。

而兼容性问题比定制代码的问题还要小。在某种程度上，定制代码是使用模板编程的一个正常部分。以一个名为 *VehicleSalesRecord* 的类为例，该类将使用链表模板工作。对于这一类，你不得不定义诸如小于(<)、等于(==)或大于(>)这样的操作，提供这些操作对于每个类都是使用模板工作的开销的一部分。

要使用容器模板，程序员通常必须定制容器中的对象，而不是容器模板本身。当需要改变模板的工作方式时，问题出现。例如，设需要定制项目的排序方法。要使用基于模板的类，就必须理解模板的初始代码，修改此原码，重新编译程序。当然这是假设有进入最初模板定义的途径(还可能没有这种途径)。但是修改了代码的模板可能不会保持其原来的功能。

总之，由于基于模板的容器类内在地要慢一些，并且模板在历史上没有形成标准，又不容易定制，所以需要有更好的方法。

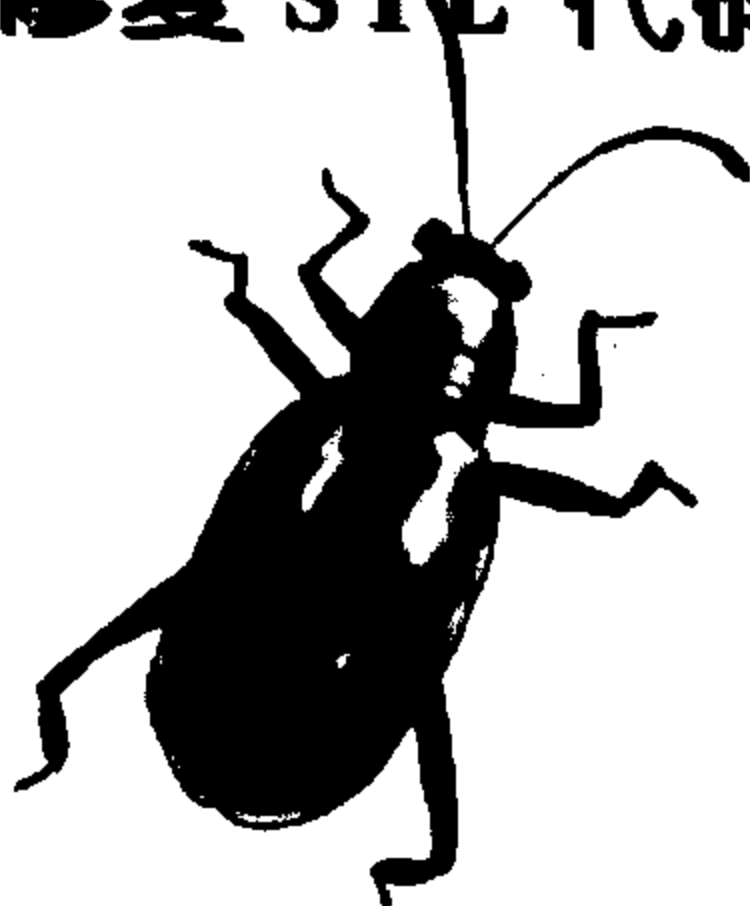
12.19 小结

如果首先学习 C 程序设计语言，然后又发展到 C++，那将会知道这两种语言有很大的差别。以类似的方式，从 C++ 发展到带有 STL 语法的 C++，可能使我们对其差别感到吃惊。本章的目的是使读者有一个高水平的飞跃，快速理解 STL。但是，本章不能代替正规的课程，也不是专门教 STL 的课本。

在下一章中，将使用以前讨论过的 Debugger 的功能来定位 STL 代码错误。但是由于 STL 错误与 STL 编程思想有关，就 C++ 与 C 相当不同的意义上说，STL 不是一种新的语言，所以对于查找特定的 STL 错误，Debugger 自身的功能没有多少帮助。 ■

第 13 章

定位、分析和
修复 STL 代码中的错误





好消息，调试 STL 代码不需要学习任何新的 Visual C++ Debugger 功能！相反，高效的 STL 调试来自于对 STL 组件之间相互关系的理解。在上一章中，介绍了 STL 的设计思想，这一思想促使 ANSI/ISO 采纳了 HP 公司对标准 C++ 语法的建议，在 C++ 中增加了 STL。

随着计算机界逐渐采纳 STL 技术，大多数程序员开始使用 STL。STL 错误最初是在由非 STL 算法向 STL 技术转变过程中出现的。在本章中，用一个典型而真实的例子。将一个非 STL 算法的程序转变为 STL 语法，为读者介绍如何调试 STL 程序。STL 错误的原因通常是不熟悉 STL 代码的要求与标准 C++ 代码实现之间的差别，将非 STL 代码转变为 STL 语法时，我们可以看到 Debugger 如何标记这种错误，然后如何将其修复。

设计提示

现在计算机界逐渐采纳可靠的 STL 技术，许多程序员第一次遇到这种新的编程思想。尽管 STL 使用标准的 C/C++ 语法，但其包含了一些与标准 C++ 语言不太一样的性质，阻挠了初学者。初始的 STL 错误多来自于对 STL 组件之间相互作用的错误理解，而与 C/C++ 语言本身无关。如果还没学习过 STL，那么花些时间学习这一有活力而功能强大的 ANSI/ISO C++ 标准新增部分是值得的！

13.1 从标准 C++ 转向 STL 语法的过程中出现的问题

由于 STL 使用标准 C++ 语法，STL 的程序错误没有与标准 C/C++ 算法不同的来源。STL 错误来源于错误使用 STL 组件。下面的讨论突出了那些最可能引起程序错误的 STL 设计思想。

13.1.1 用迭代器遍历容器

STL 和所有其他 C++ 容器类库之间最重要的区别在于大多数 STL 算法是类属的，它们作用于各种容器，甚至作用于普通的 C++ 数组。STL 库设计的一个关键因素是始终把迭代器作为算法和容器之间的结合物使用，可以把迭代器看作推广了的指针。在上一章中，学习了迭代器可以精确地分为五类，这是决定哪些算法可以与哪些容器一起使用的基础。对于扩充 STL 库，迭代器也是一个主要向导，这些库包括与 STL 容器一起使用的一些新的算法，或许多 STL 类属算法适用的一些新的容器。

24x7

记住要始终使用 STL 迭代器遍历容器，要使用特定的 STL 方法，例如 `begin()` 和 `end()` 定位开头和结尾的容器元素。

13.1.2 仔细研究迭代器

容器本身不提供对其元素的访问，而迭代器被用来遍历容器中的元素。迭代器非常类似于灵活的指针，有增量和取内容(*)操作。通过由迭代器统一对容器的访问，STL 以一致的方式与容器相互作用成为可能。另外，迭代器是将算法和容器连在一起的黏合剂。

迭代器是 STL 设计的基础，它给了 STL 最大的灵活性。算法不是为特定的容器开发的，而是为特定的迭代器类别开发的。这一思想使同样的算法能够用于各种不同的容器。

每一类别的迭代器有一组要求，这组要求必须分别由该类别迭代器的具体类型满足。一个给定迭代器类别的要求是由一组该种迭代器的合法表达式加上描述其用途的准确语义指定的，另外 STL 中的迭代器必须满足复杂性要求，这些要求保证根据迭代器写出的算法能够正确而高效地工作。

STL 提供了迭代器类别的一个分级结构(见第 12 章)。最上层的迭代器通用性最强，最底层的受到最多的限制并且有最少的需求。迭代器满足比它低级的迭代器的所有要求，每个容器都限定了其自身属于哪个迭代器类别。通过使用一种新颖的语言技术，STL 在编译时根据迭代器的类别选择正确的算法。

迭代器使用的方式类似于指针。正如指针那样，对于任何类型的迭代器，保证有一个迭代器值指向对应容器的最后一个元素的后面。每个 STL 容器提供成员函数 *begin()* 和 *end()*，它们为第一个和结尾(最后一个的后面一个)元素返回迭代器值。

下面进行一个迭代操作，遍历一个 `int` 列表：

```
list::iterator it;
for (it = myList.begin(); it != myList.end(); it++;)
{
    cout << *myList << endl;
}
```

注意每次通过循环，对迭代器的取内容操作都得到了迭代器指向的值，增量运算符用来推动迭代器遍历整个列表。

STL 还提供常量(`const`)迭代器，这样的迭代器可以与 `const` 容器一起使用：

```
template <class T>
void OutputList (const list<T>& myList)
{
    list::const_iterator it;
    for (it = myList.begin();
        it != myList.end();
        it++;) {
        cout << *myList << endl;
    }
}
```



```
}  
}
```

13.1.3 流迭代器

C++标准库提供了输入/输出流(iostreams)以便于从输入流读取数据和从输出流写出数据。STL 提供了两个迭代器模板, 以使算法可以直接操作 I/O 流:

istream_iterator, 从一个输入流读数据

ostream_iterator, 写数据到一个输出流

例如, 可以如下面那样将数据从标准输入设备读入列表:

```
istream_iterator<int, ptrdiff_t> in(cin);  
istream_iterator<int, ptrdiff_t> eos;  
copy (in, eos, back_inserter(l));
```

然后将此列表写出到标准输出设备:

```
ostream_iterator out(cout, ",");  
copy (myList.begin(), myList.end(), out);
```

13.1.4 为什么使用 end()

对于任何 STL 容器, *iterator::end()*指向容器中最后一项的后面一个单元, 因此是一个“非法指针”; *iterator::end()*不是指向容器中的最后一项, 而指向这样一个单元: 如果使用 *push_back*, 那么下一项将从此处进入容器。

为什么 *iterator::end()*指向容器末尾的后面? 为什么不指向容器的最后一项? 原因相当简单: STL 容器使用 C 指针语义, 而 *end()*的返回值与 C 的 NULL 指针等价。考虑如果 *end()*改为返回容器的最后一项, 你将如何继续处理:

```
MyMap::const_iterator it = my_dataBase.find(key);  
if (it == my_dataBase.end())  
    no_match_key();  
else  
    match_key_found();
```

或者:

```
bool empty(const STL_Container& containerInstance) {  
    return containerInstance.begin() == containerInstance.end();  
}
```

在 STL 中, 永远记住 *begin()*返回容器的第一项(如果存在的话), 而 *end()*返回容器末尾

的后面一项。

13.1.5 复制列表

使用 STL 的 *copy()*、*remove()* 或任何其他试图将一个序列复制到另一个序列的算法时，要确保原容器至少和结果容器一样大。当从一个有内容的列表复制到一个新定义的尚无内容的列表时，不得不更加小心。例如，查看下面的代码段：

```
list<int> list_A;
list<int> list_B;
copy(list_A.begin(), list_A.end(), list_B.begin()); //Failed code.
```

当正确地使用 STL 重写时

```
copy(list_A.begin(), list_A.end(), back_list_iterator<int> (list_B));
```

这条语句可行是因为 *back_list_iterator* 调用了根据需要动态地调整列表大小的 *push_back()*。

错误监视

当复制容器时，要确保原容器至少和结果容器可能变成的一样大。

13.1.6 列表中的列表

经常，一个 STL 列表的元素还是列表。遍历这样复杂的容器实际上是 STL 技术的简单再利用。下面的代码段说明了如何实现这一算法：

```
#include <list>
#include <iostream>
using namespace std;
typedef list<int> ListElement;
typedef list<ListElement> OwningList;
void outputList(const ListElement& EntireListElement, int listNumber) {
    ostream_iterator<int> out(cout, " ");
    cout << "list " << listNumber << ": ";
    copy(EntireListElement.begin(), EntireListElement.end(), out);
    cout << endl;
}
void main ( void ) {
    OwningList listOfListElements;
    // initialize listOfList: total of 3 lists, each with 5 members.
    For(int i = 0; i < 3; ++i) {
        ListElement EntireListElement;
```



```

        for(int j = 0; j < 5; ++j) {
            EntireListElement.push_back(i * 4 + j);
        }
        outputList(EntireListElement, i+1);
        listOfListElements.push_back(EntireListElement);
    }
    cout << endl;
    // traversing entire list with list elements.
    OwingingList::iterator it = listOfListElements.begin();
    For(int j = 1; it != listOfListElements.end(); ++it, ++j) {
        Const ListElement& EntireListElement = *it;
        OutputList(EntireListElement, j);
    }
}

```

这一简单的 STL 算法把每个 *OwingingList* 元素作为一个独立的列表处理。无论何时，只要没有任何 *OwingingList* 列表或者 *it!=listOfListElements.end()*，最后一个 for 循环将停止迭代并跳到 *outputList()* 将列表枚举输出！

13.1.7 STL 字符串指针的麻烦

下面的例子将说明，保存指向 STL 容器内容的指针有另一个讨厌的副作用。首先，*list<char*>* 的声明定义了一个字符型指针的列表，而不是一个其指向的字符串的列表。因此 STL 的 *less<char*>* 将比较 *char** (指针)，而非 C++ 字符串类：

```

char buf[1024];
strcpy(buf, "Test String");
list<char*> list_A;
list_A.push_back(buf);
ostream_iterator<char*> citer(cout, " ");
copy(list_A.begin(), list_A.end(), citer);
// you should see one string "Test String"
strcpy(buf, "NewString");
copy(list_A.begin(), list_A.end(), citer);
// The list changed and it should not have!

```

24x7

要遍历其每个元素也是列表的列表，只需把每个列表元素看作一个独立的列表。这类似于处理一个结构数组，处理这一结构数组时每一个元素如同处理一个独立的结构，而不是一个容器的一部分。

总的来说，不要把 *char** 当作容器对象使用，而应当把列表元素作为 C++ 字符串类实现，比如：


```
typedef list<string> sringList;
```

错误监视

尽管能够在语法上创建一个 *char** 容器，但这有潜在的危险，因为 STL 特定的方法可能将其算法不正确地应用于数据的地址，而不是数据本身！

13.1.8 释放 STL 指针

如果创建了指针容器，一定要在代码中明确地释放存储空间，尤其是如果容器建立在堆栈上并超出堆栈的范围产生一个内存泄漏的话。STL 容器复制和删除的只是保存指针所需的存储空间，而不是指针指向的对象。程序员可以创建模板化的删除程序，类似于下面：

```
template <class FwdIt, class FwdIt>
void sequence_delete(FwdIt first, FwdIt last) {
    while (first != last)
        delete *first++;
}
template <class FwdIt, class FwdIt>
void map_delete(FwdIt first, FwdIt last) {
    while (first != last)
        delete (*first++).second;
}
Map<int, DataType*, less<int> > myMap;
// properly releases myMap pointers.
map_delete(myMap.begin(), myMap.end());
```

对 C++ 函数 `delete()` 的正式调用完成了彻底的返回：

```
delete (*first++).second
```

此函数的参数是被引用的对象指针 **first*。

设计提示

使用动态内存分配时，要确保在程序中某个地方使用了函数 `delete()`，以将内存正确恢复为可用内存。

13.2 一个 C++ 程序转变为 STL 语法的例子

在本节中，将把一个实用的标准 C++ 面向对象的程序转换为 STL 语法形式。程序实现了一个常见的称为 War 的纸牌游戏。代码转换是逐步执行的，如同一个刚学习 STL 的程序



员要做的那样。修改每一处代码段后立即跟着一个建立和调试的过程，以检验新的 STL 程序的正确性，同时遇到一些没有预料到的错误。首先，游戏的标准 C++ 面向对象的程序是：

```
//
// wargame.cpp
// Learning how to convert a standard C++ code solution
// into a robust STL design
// Chris H. Pappas and William H. Murray, 2000
//
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;
typedef enum tagSuits {diamond, club, heart, spade} SUITS;
//*****
// class aSingleCard with defined methods
class aSingleCard {
public:
    int iRank;
    SUITS suit;
    aSingleCard() {iRank = 0; suit = spade;};
    aSingleCard( SUITS s, int ir) {suit = s; iRank = ir;};
    friend ostream& operator <<( ostream& out, aSingleCard& aCard );
};
//*****
// class WarDeck with defined methods
class WarDeck {
public:
    WarDeck();
    void shuffleDeck();
    bool isDeckEmpty();
    aSingleCard drawCardFromDeck();
protected:
    #define DECK_SIZE      52
    #define CARDS_PER_SUIT 13
    aSingleCard fullDeck[DECK_SIZE];
    int iCurrentCard;
};
WarDeck::WarDeck()
{
    iCurrentCard = 0;
```

```

for( int i = 1; i <= CARDS_PER_SUIT; i++ ) {
    aSingleCard c1(diamond, i), c2(spade, i), c3(heart, i), c4(club, i);
    fullDeck[iCurrentCard++] = c1;
    fullDeck[iCurrentCard++] = c2;
    fullDeck[iCurrentCard++] = c3;
    fullDeck[iCurrentCard++] = c4;
}
}
void WarDeck::shuffleDeck()
{
    random_shuffle( fullDeck, fullDeck+52);
}
aSingleCard WarDeck::drawCardFromDeck()
{
    if( ! isDeckEmpty() )
        return fullDeck[--iCurrentCard];
    else {
        aSingleCard defaultCard( spade, 1 );
        return defaultCard;
    }
}
bool WarDeck::isDeckEmpty()
{
    return iCurrentCard <= 0;
}
//*****
// class Opponent with defined methods
class Opponent {
public:
    Opponent( WarDeck& );
    aSingleCard drawCardFromDeck();
    void addUpPoints( int );
    int what_sTheScore();
    void putBackCard( WarDeck& );
protected:
    #define CARDS_IN_HAND 3
    aSingleCard currentHand[CARDS_IN_HAND];
    int currentScore;
    int cardBeingPlayed;
};
Opponent::Opponent( WarDeck & aDeck )
{
    currentScore = 0;
}
    
```



```
        for( int i = 0; i < CARDS_IN_HAND; i++ )
            currentHand[i] = aDeck.drawCardFromDeck();
        cardBeingPlayed = 0;
    }
    aSingleCard Opponent::drawCardFromDeck()
    {
        cardBeingPlayed = rand() % 3;
        return currentHand[cardBeingPlayed];
    }
    void Opponent::addUpPoints( int howMany )
    {
        currentScore += howMany;
    }
    int Opponent::what_sTheScore()
    {
        return currentScore;
    }
    void Opponent::putBackCard(WarDeck& aDeck)
    {
        currentHand[cardBeingPlayed] = aDeck.drawCardFromDeck();
    }
    //*****
    // main() function
    void main( void )
    {
        WarDeck actualDeck;
        actualDeck.shuffleDeck();
        srand( (unsigned int)time( NULL ) );
        Opponent opponent1(actualDeck);
        Opponent opponent2(actualDeck);
        while( !actualDeck.isDeckEmpty() ) {
            aSingleCard card1 = opponent1.drawCardFromDeck();
            cout << "Opponent 1 plays " << card1 << endl;
            aSingleCard card2 = opponent2.drawCardFromDeck();
            cout << "Opponent 2 plays " << card2 << endl;

            if( card1.iRank == card2.iRank ) {
                opponent1.addUpPoints(1);
                opponent2.addUpPoints(1);
                cout << "Players tie\n" << endl;
            }
            else if( card1.iRank > card2.iRank ) {
                opponent1.addUpPoints(2);
            }
        }
    }
}
```

```

        cout << "Opponent 1 wins round\n";
    }
    else {
        opponent2.addUpPoints(2);
        cout << "Opponent 2 wins round\n";
    }
    opponent1.putBackCard(actualDeck);
    opponent2.putBackCard(actualDeck);
    cout << "\n\nPress ENTER to continue." << endl;
    cin.get();
}
cout << "Opponent 1 what_sTheScore " << opponent1.what_sTheScore() << endl;
cout << "Opponent 2 what_sTheScore " << opponent2.what_sTheScore() << endl;
}
//*****
// class aSingleCard friend overloaded insertion operator
ostream& operator << (ostream& out, aSingleCard& aCard)
{
    switch( aCard.iRank ) {
        case 1: out << "Ace" ; break;
        case 11: out << "Jack" ; break;
        case 12: out << "Queen"; break;
        case 13: out << "King" ; break;
        default:
            out << aCard.iRank; break;
    }
    switch( aCard.suit ) {
        case diamond: out << " of Diamonds"; break;
        case spade : out << " of Spades "; break;
        case heart : out << " of Hearts "; break;
        case club : out << " of Clubs "; break;
    }
    return out;
};

```

13.2.1 第一步——更新 aSingleCard 类

aSingleCard 类包含两个独立的数据成员，*iRank* 和 *suit*。在 *wargame.cpp* 中，这两个成员始终在逻辑上相互联系，分别表示每张牌的数字和花色。在转向 STL 设计时，不希望用两个向量分别表示数字和花色，而是需要一个向量包含所有 52 张牌，作为单个实体表示。

在 STL 设计语法中，两个逻辑上相互联系的数据成员作为 STL 对(pair)正式保存。使 *iRank*



和 *suit* 成为对是通过下面的 **typedef** 语句完成的:

```
typedef struct pair<int,SUITS> ACARD_PAIR;
```

当然,这一新的 **typedef** 要求更新 *aSingleCard* 类:

```
/*******  
// STL pair data type containing int and SUITS  
typedef struct pair<int,SUITS> ACARD_PAIR;  
/*******  
// class aSingleCard with defined methods  
class aSingleCard {  
    public:  
        // non-STL int iRank;  
        // non-STL SUITS suit;  
        // STL replacement  
        ACARD_PAIR oneCard;  
        aSingleCard() {iRank = 0; suit = spade;};  
        aSingleCard(SUITS s, int ir) {suit = s; iRank = ir;};  
        friend ostream& operator << ( ostream& out, aSingleCard& aCard );  
};
```

首先尝试建立可执行程序,产生了下列调试错误信息(参见图 13-1),标记“未声明标识符” *iRank* 和 *suit*。

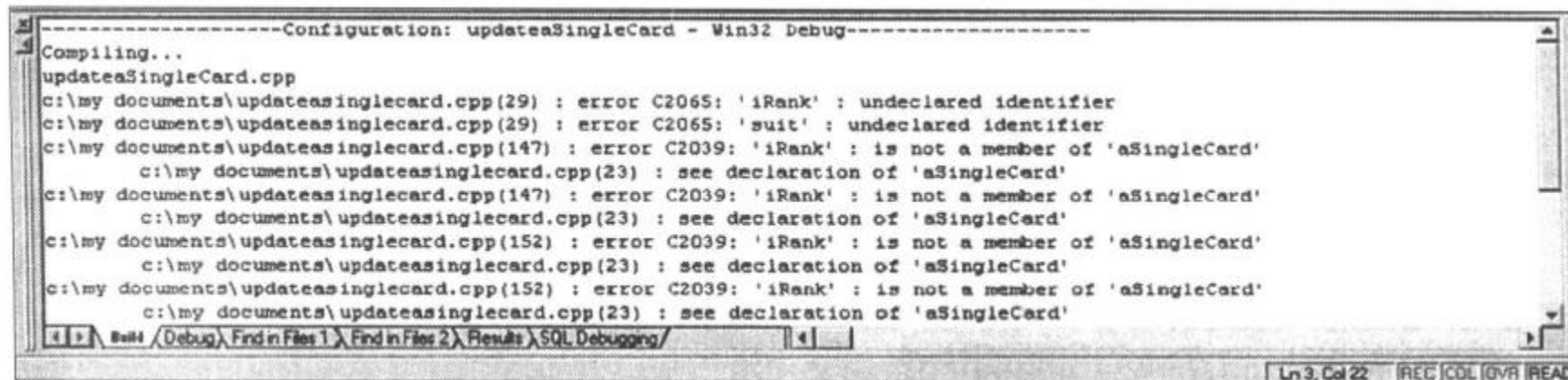


图 13-1 STL 对(pair)的问题

由于 STL 对有两个预定义的数据成员, *first* 和 *second*, 所以 *aSingleCard* 类中每个对 *iRank* 的引用必须转向 **pair** 成员名字 *first*。每个对 *suit* 的引用改变为 **pair** 成员名字 *second*, 例如:

```
/*******  
// class aSingleCard with defined methods  
class aSingleCard {  
    public:  
        // non-STL int iRank;
```

```
// non-STL SUITS suit;
// STL replacement
ACARD_PAIR oneCard;
aSingleCard() {oneCard.first = 0; oneCard.second = spade;};
aSingleCard(SUITS s, int ir) {oneCard.second = s; oneCard.first = ir;};
friend ostream& operator <<( ostream& out, aSingleCard& aCard );
};
```

更新为 **pair** 成员名字 *first* 和 *second* 之后, 又一次尝试建立可执行程序, 不料在 **main()** 中竟发现更多的对 *iRank* 和 *suit* 的引用(参见图 13-2)。

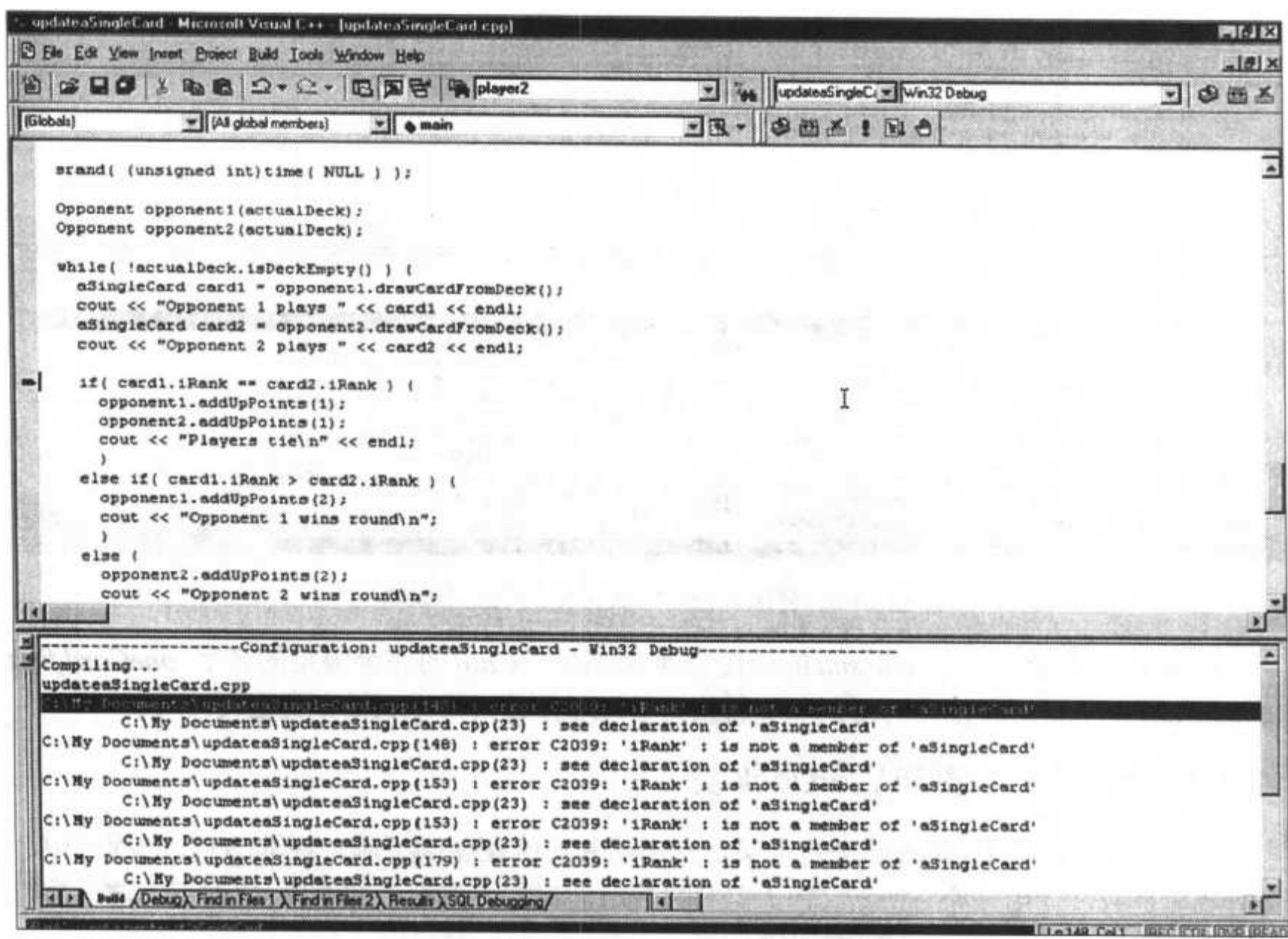


图 13-2 检测所有对 *iRank* 和 *suit* 的非法引用

迅速使用 Visual C++ 的 Find 和 Replace 选项定位每一处对原来数据成员名字(*iRank* 和 *suit*) 的错误使用, 并将其修改为 **pair** 成员名字 *first* 和 *second*。然而, 再次尝试建立可执行程序, 仍然没有成功(参见图 13-3)。

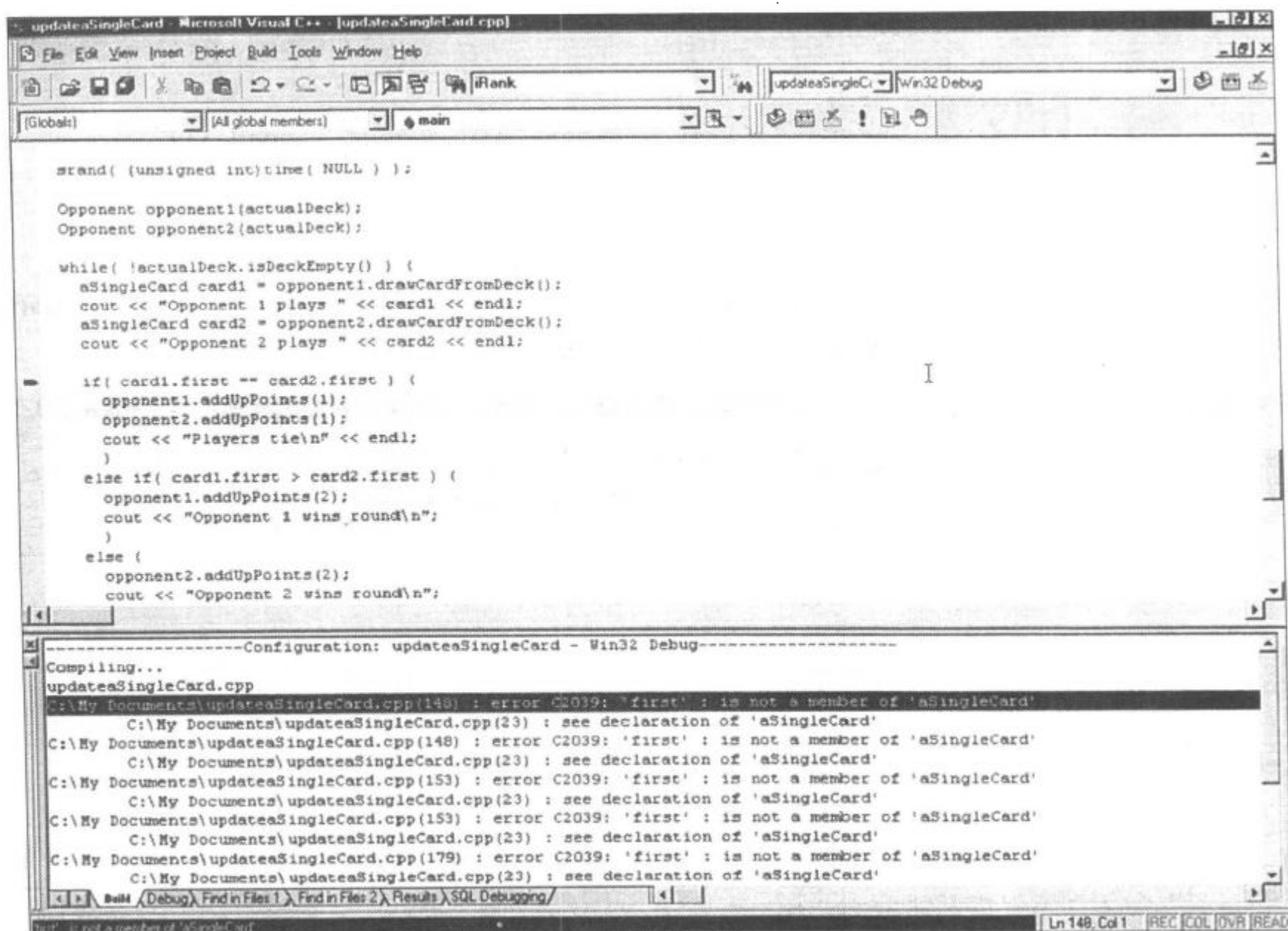


图 13-3 替换为 first 和 second 后仍然不成功

实际上,需要将*iRank*和*suit*的引用替换为*aSingleCard*的完全限定的名字:*oneCard.first*和*oneCard.second*。现在我们得到一个转换为 STL 语法的 *aSingleCard* 类定义。

13.2.2 第二步——更新 WarDeck 类

wargame.cpp 对 WarDeck 类的定义负责声明和实例化整副纸牌。在非 STL 的程序中,一副纸牌是通过使用一个 *aSingleCard* 实例的数组创建的。然而,在 STL 技术下,这一数组需要通过使用 STL **vector**(向量)类创建。

代码转换的第一步需要一个 **typedef** 定义(**typedef** 定义符合整个行业标准,而不是 STL 单独需要的)声明一个新的类型,例如:

```
typedef vector<aSingleCard> CardVector;
```

声明一个新的类型尽管花一些时间,但再次使用应用程序特定的这种定义将变得容易。

这一行业标准以前也被 *ACARD_PAIR* 的 `typedef` 使用。老的和新的 *WarDeck* 类定义之间的最初的代码变化涉及到两个语句的简单替换：

```
aSingleCard fullDeck[DECK_SIZE];
```

替换为：

```
CardVector fullDeck;
```

注意在更新后的代码中不见了的 *DECK_SIZE*。STL 向量是动态的，与标准 C++ 数组那样需要定义初始大小不同。然而在接下来的一次建立期间，又发现了几个令人灰心的错误信息，如图 13-4 所示。



图 13-4 什么语句缺少了分号

这时 Debugger 标记缺少一个分号。检查被标记语句(`typedef vector...` —— 参见图 13-4)之前的所有语句，我们被搞糊涂了，似乎没有缺少分号！这是转向 STL 引起的问题。尽管代码的确已经修改了必要的组件，以在语法上声明一个 `vector typedef` 和更新 *WarDeck* 的 *fullDeck* 的数据类型(由数组变为向量)；但是，算法忘记了包含定义 `vector` 所必需的 STL 头



文件。包含的头文件列表需要由：

```
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <ctime>
```

改为：

```
#include <vector> //added for vector definitions
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <ctime>
```

更新#include 语句后，再次建立可执行程序，Debugger 又标记了另一个问题(参见图 13-5)。

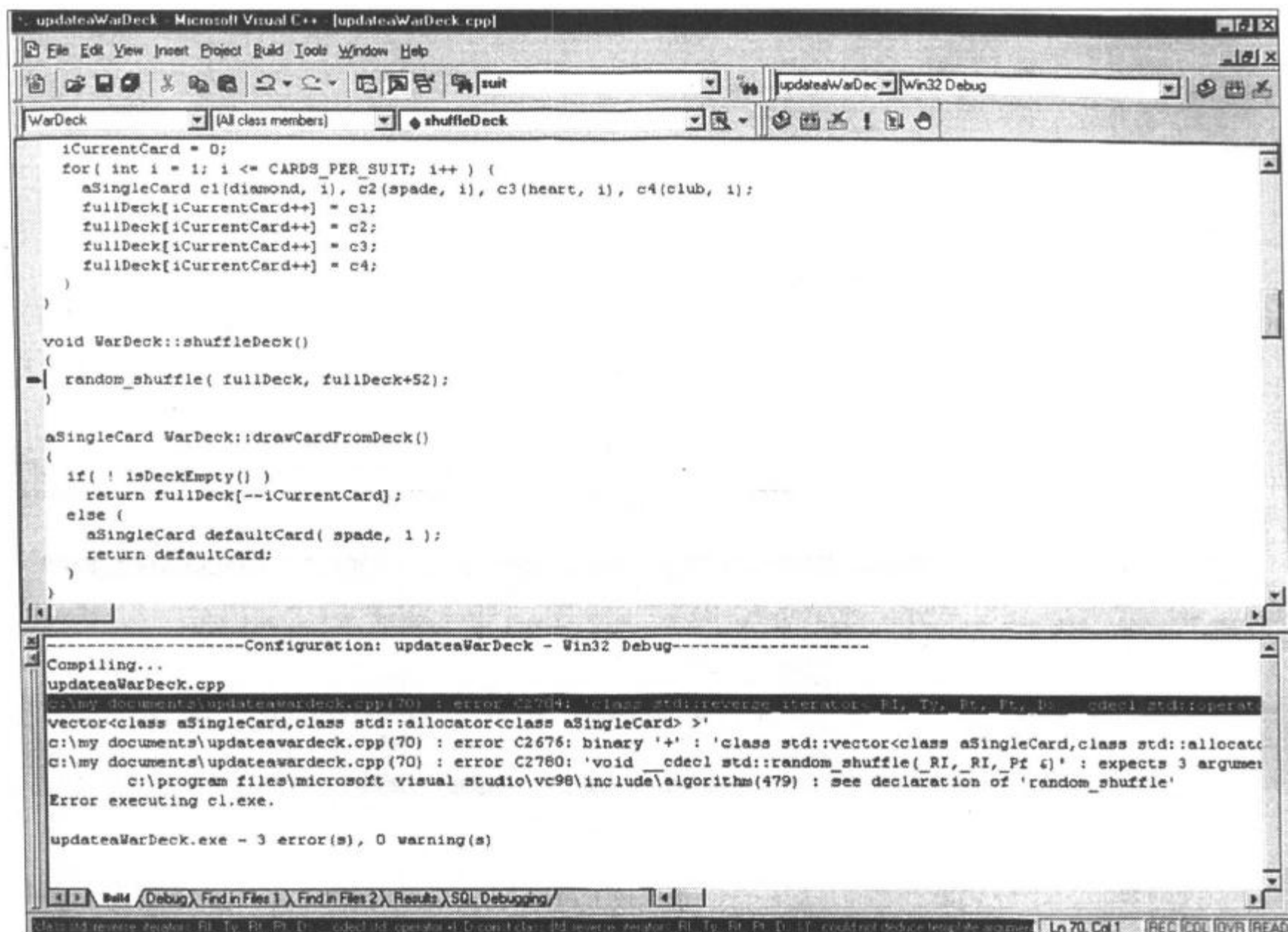


图 13-5 STL 错误代码语法

仔细检查 Debugger 的第一条错误令人头痛的信息:

```
c:\chapter13\updateawarddeck.cpp(70) : error C2784: 'class
std::reverse_iterator<RI, _Ty, _Rt, _Pt, _D> _cdecl
std::operator +(_D, const class std::reverse_iterator<_RI,
_Ty, _Rt, _Pt, _D> &)' : could not deduce template argument for '' from 'class
std::
```

除非很好地训练过调试 STL 语法, 否则这条信息的语法看起来是那么复杂, 以致没有下一步在何处排除这一错误的线索。

再一次, 对 STL 错误的解决牵涉到对 STL 组件之间相互作用和语法要求的理解。最初的 *WarDeck::shuffleDeck()* 方法使用数组的名字 *fullDeck* 将整副牌中第一张牌的实际起始地址传递给函数 *random_shuffle()*, *random_shuffle()* 的第二个参数是计算出的, 相对 *fullDeck* 跳过 52 个元素, 即 *fullDeck+52*:

```
void WarDeck::shuffleDeck()
{
    random_shuffle(fullDeck, fullDeck+52);
}
```

然而, STL 不接受这种语法。相反, STL 在语法上要求使用一个合适的迭代器, 也就是推广了的指针。STL 语法需要首先用一个迭代器指向一个特定的类型——在本例中为 *CardVector* 元素, 这是通过以下语句完成的:

```
typedef CardVector::iterator CardVectorIt;
```

再次建议首先用 **typedef** 声明一个新类型。迭代器的名字 *CardVectorIt* 不是随便命名的。“*CardVector*”部分将标识符的名字与数据类型的定义联系在一起, 而 STL 语法在技术上不需要, 但 STL 程序员一致同意的 “It” 部分, 表明了这一标识符的类型是迭代器 (Iterator)。

然而, 新的 **typedef** 不充分。所以使用 STL 指定的容器类方法 *begin()* 和 *end()* 完成了对 **vector** 开始和结尾元素的地址的引用, 由这些方法返回的地址保存在正确定义的迭代器变量中, 例如:

```
CardVectorIt Start, End;
```

整个 *WarDeck::shuffleDeck()* 方法现在重写为:

```
void WarDeck::shuffleDeck()
{
    typedef CardVector::iterator CardVectorIt;
    CardVectorIt Start, End;
    Start = fullDeck.begin();
    End = fullDeck.end();
```



```
    random_shuffle( Start, End);  
}
```

注意方法 `begin()` 和 `end()` 是如何被 `CardVector` 类型的实例 `fullDeck` 继承的, 程序成功地完成了一次 `random_shuffle()` (洗牌) 并满足正确的 STL 语法要求: 两个实际参数 `Start` 和 `End` 是迭代器类型的。

执行前面的代码修改, 再次建立可执行程序并且成功了——“0 error(s), 0 warning(s)”。确认转换成功之后, 执行程序。然而, 图 13-6 却显示了另一种情况。

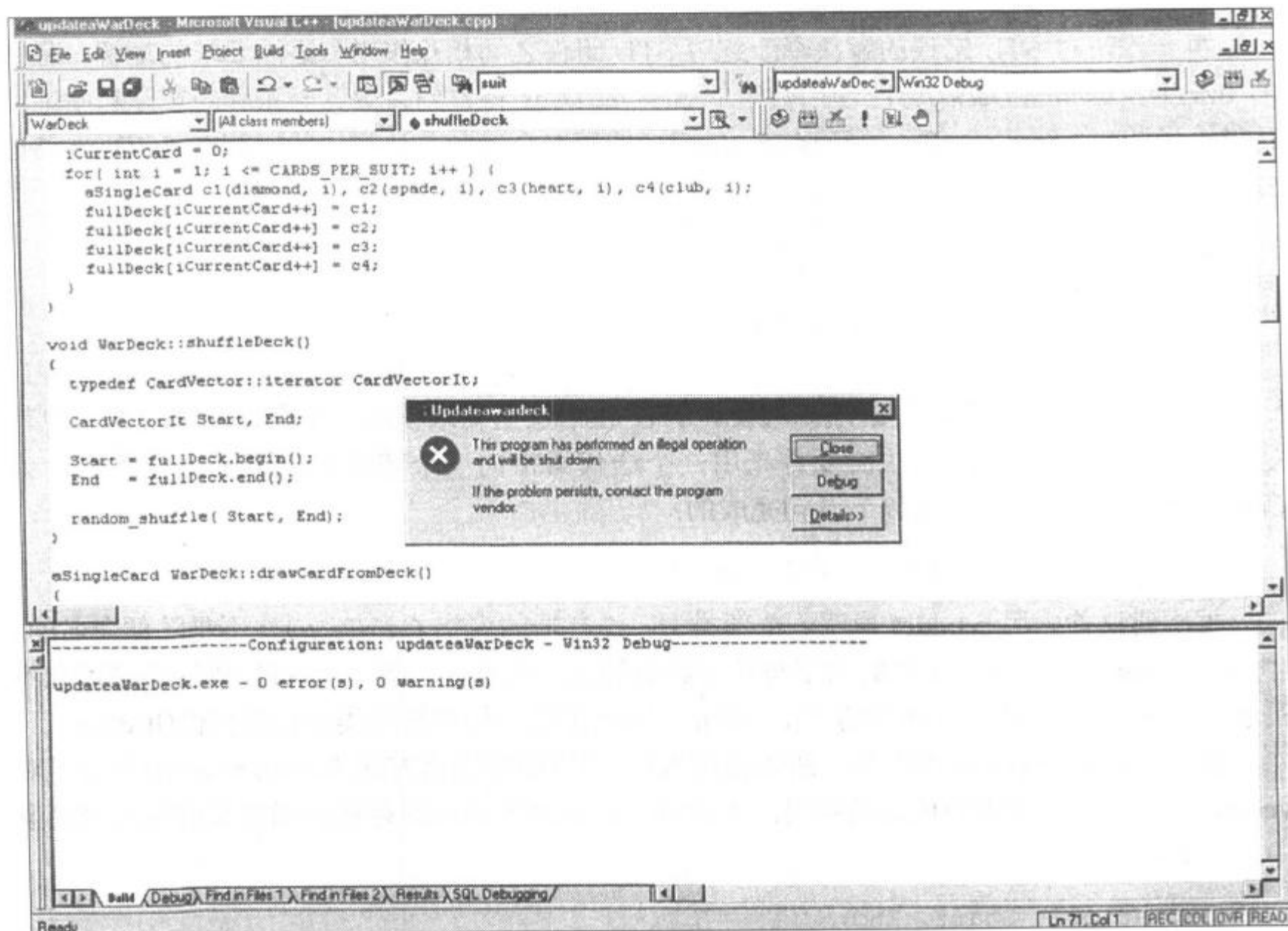


图 13-6 更多的问题

最糟的噩梦发生了——*This program has performed an illegal operation and will be shut down* (程序执行了非法操作, 将被关闭)。下面应该如何处理?

13.2.3 第三步——修复 STL 代码的执行错误

Debugger 中没有一个这样工具, 可以随意使用并且熟练地解释 STL 代码的执行错误。

这次的运行时错误典型地是由于错误使用 STL 组件而发生的。

尽管 *WarDeck* 类已经由数组语法转变为向量语法并且方法 *shuffleDeck()* 也被更新了，但是算法中仍然存在非 STL 语法和 STL 语法之间的不一致。查看最初的 C++ 方法 *drawCardFromDeck()*:

```
aSingleCard WarDeck::drawCardFromDeck()
{
    if( !isDeckEmpty() )
        return fullDeck[--iCurrentCard];
    else {
        aSingleCard defaultCard( spade, 1 );
        return defaultCard;
    }
}
```

对 *WarDeck::isDeckEmpty()* 的调用看起来没有什么错误，但实际上包含了一个 STL 设计缺陷。首先，最初的 C++ 语法的 *isDeckEmpty()* 版本是:

```
bool WarDeck::isDeckEmpty()
{
    return iCurrentCard <= 0;
}
```

现在，在 STL 语法中它应当写为:

```
bool WarDeck::isDeckEmpty()
{
    // non-STL return iCurrentCard <= 0;
    return fullDeck.empty();
}
```

动态分配 STL 向量容器时，找到向量开头和结尾地址的最好途径是使用一个 STL 容器指定的方法——在本例中为 *empty()*，这是一个相对不费事的转变。那么现在方法 *drawDeckFromCard()* 应该如何处理？

drawDeckFromCard() 对应的 STL 语法形式是:

```
aSingleCard WarDeck::drawCardFromDeck()
{
    aSingleCard tempCard;
    if( !isDeckEmpty() ) {
        tempCard = fullDeck.front();
        fullDeck.erase(fullDeck.begin());
    }
    else {
        aSingleCard defaultCard( spade, 1 );
        tempCard = defaultCard;
    }
}
```



```
    }  
    return tempCard;  
}
```

转换后的算法需要为抽出的纸牌声明一个临时夹具 *tempCard*。下一步，更新后的 *if* 测试条件使用 STL *empty()* 方法来检测一副牌是否为空。STL 方法 *front()* 返回整副牌中第一张牌的地址。对 STL 方法 *begin()* 的调用返回一个指针，指向从整副牌 *fullDeck* 中拿走的纸牌。该指针用作 STL 继承的方法 *erase()* 的实际参数，从整副牌中除去该张纸牌。

13.2.4 第四步——更新 Opponent 类

如果还记得在最初的代码中所有 C++ 数组引用必须转换成相应的 STL 向量，那么 *Opponent* 类最初的转换看起来就简单了。为了使引用变得容易，最初的 *Opponent* 类定义形式如下：

```
class Opponent {  
public:  
    Opponent( WarDeck& );  
    aSingleCard drawCardFromDeck();  
    void addUpPoints( int );  
    int what_sTheScore();  
    void putBackCard( WarDeck& );  
protected:  
    #define CARDS_IN_HAND 3  
    aSingleCard currentHand[CARDS_IN_HAND];  
    int currentScore;  
    int cardBeingPlayed;  
};
```

再一次，唯一需要改变的类定义是数组定义：

```
aSingleCard currentHand[CARDS_IN_HAND];
```

将其改为：

```
CardVector currentHand; // STL dynamic vector syntax
```

余下的代码修改是 STL 设计方法特定的，举 *Opponent* 构造函数为例：

```
Opponent::Opponent( WarDeck & aDeck )  
{  
    currentScore = 0;  
    for( int i = 0; i < CARDS_IN_HAND; i++ )  
        currentHand[i] = aDeck..drawCardFromDeck();  
    cardBeingPlayed = 0;  
}
```

```

    }

```

使用 STL 语法应修改为:

```

Opponent::Opponent( WarDeck & aDeck )
{
    currentScore = 0;
    for( int i = 0; i < CARDS_IN_HAND; i++ )
        // non-STL currentHand[i] = aDeck.drawCardFromDeck();
        currentHand.push_back(aDeck.drawCardFromDeck());
    cardBeingPlayed = 0;
}

```

在 STL 技术下, 使用方法 *push_back()* 为向量添加一个元素, 注意在对应的非 STL 程序中删除任何有数组下标的语句:

```

currentHand.push_back(aDeck..drawCardFromDeck());

```

13.2.5 第五步——运转的 STL 程序

前面做完了所有必要的代码修改并且成功地建立了可执行程序, 现在执行这一程序, 然而却再次失败了。由于 STL 代码调试不需要任何隐藏的 Visual C++ Debugger 功能, 所以现在依次使用 F10(跳过)和 F11(跳入)命令。最后发现致命的异常出现在构造函数 *WarDeck()* 中(参见图 13-7)。

注意对错误出现之处诊断只需标准的 Debugger 命令, 如跳入和跳过, 然而解决代码的错误需要有 STL 组件的专门知识。该出错的构造函数保留了数组特定的语法和引用, 不符合 STL **vector** 类的语法要求。STL 允许的构造函数形式为:

```

WarDeck::WarDeck()
{
    iCurrentCard = 0;
    for( int i = 1; i <= CARDS_PER_SUIT; i++ ) {
        aSingleCard c1(diamond, i), c2(spade, i), c3(heart, i), c4(club, i);
        // non-STL fullDeck[iCurrentCard++] = c1;
        // non-STL fullDeck[iCurrentCard++] = c2;
        // non-STL fullDeck[iCurrentCard++] = c3;
        // non-STL fullDeck[iCurrentCard++] = c4;
        fullDeck.push_back( c1 );
        fullDeck.push_back( c2 );
        fullDeck.push_back( c3 );
        fullDeck.push_back( c4 );
    }
}

```

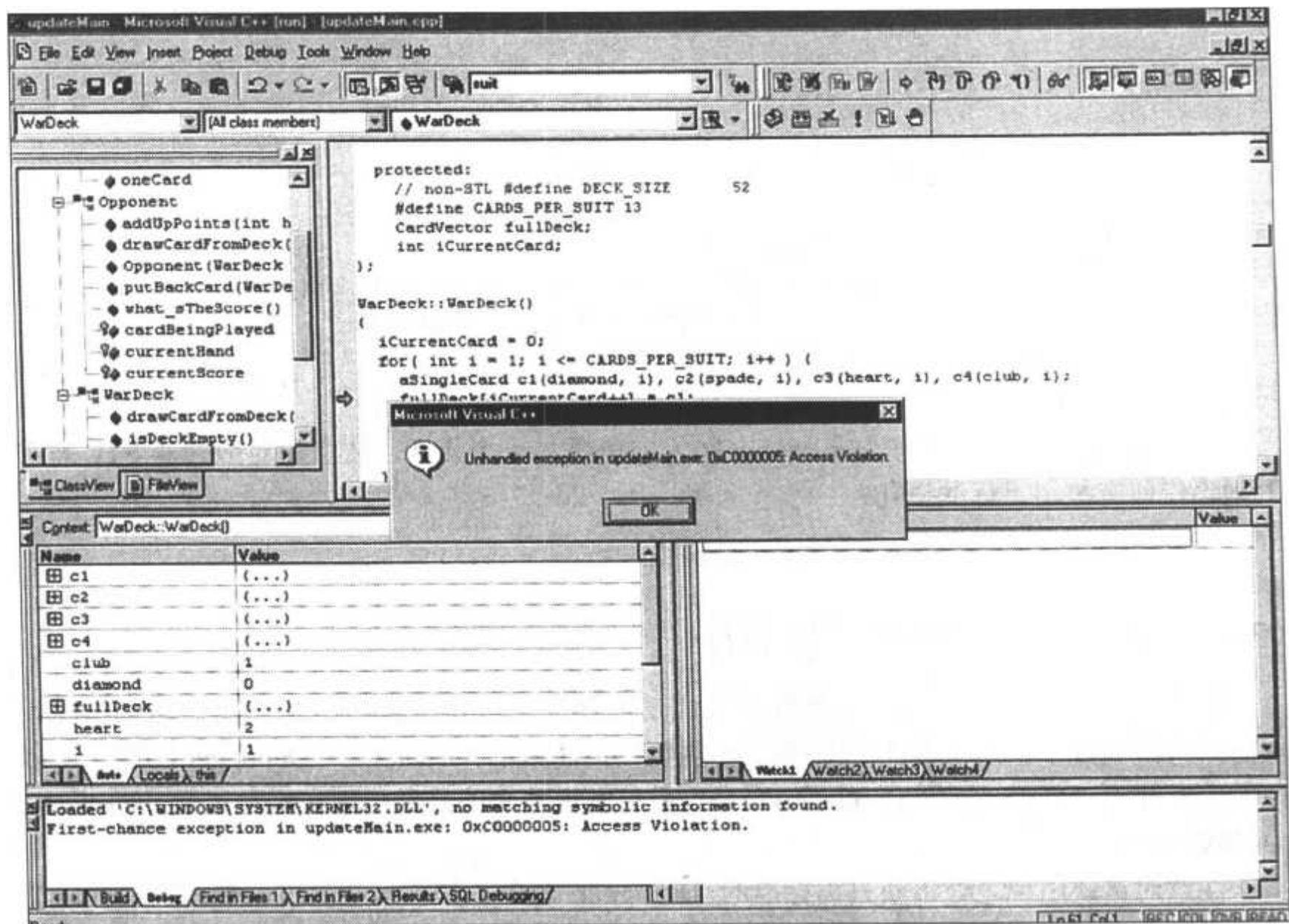


图 13-7 Debugger 标记了剩余的非 STL 代码

注意 `fullDeck` 继承的 STL `vector` 类的方法 `vector::push_back()` 用来按照花色将牌插入一副牌中。

13.3 STL 语法的源文件 `wargame.cpp`

因为对 `wargame.cpp` 做了大量的 STL 特定的转换，所以本章以转换后的源文件 `wargame.cpp` 整个清单结束。在本章的例子中夹杂着从非 STL 语法到 STL 语法的代码转换过程中产生的最典型的错误。应当把此例作为一个关于如何将应用程序转换为 STL 语法的提示。另外，此例突出了那些我们应当学习和掌握的 STL 设计思想。

```
//  
// wargame.cpp  
// Learning how to convert a standard C++ code solution
```



```
// into a robust STL design
// Chris H. Pappas and William H. Murray, 2000
//
#include <vector> // added for vector definitions
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;
typedef enum tagSuits {diamond, club, heart, spade} SUITS;
//*****
//STL pair data type containing int and SUITS
typedef struct pair<int,SUITS> ACARD_PAIR;
//*****
// class aSingleCard with defined methods
class aSingleCard {
public:
    // non-STL int first;
    // non-STL SUITS suit;
    // STL replacement
    ACARD_PAIR oneCard;
    aSingleCard() {oneCard.first = 0; oneCard.second = spade;};
    aSingleCard( SUITS s, int ir) {oneCard.second = s;
                                   oneCard.first = ir;};
    friend ostream& operator <<( ostream& out, aSingleCard& aCard );
};
//*****
// STL standard typedef for application-specific vectors
typedef vector<aSingleCard> CardVector;
//*****
// class WarDeck with defined methods
class WarDeck {
public:
    WarDeck();
    void shuffleDeck();
    bool isDeckEmpty();
    aSingleCard drawCardFromDeck();
protected:
    // non-STL #define DECK_SIZE      52
    #define CARDS_PER_SUIT 13
    CardVector fullDeck;
    int iCurrentCard;
};
```



```
WarDeck::WarDeck()
{
    iCurrentCard = 0;
    for( int i = 1; i <= CARDS_PER_SUIT; i++ ) {
        aSingleCard c1(diamond, i), c2(spade, i), c3(heart, i), c4(club, i);
        // non-STL fullDeck[iCurrentCard++] = c1;
        // non-STL fullDeck[iCurrentCard++] = c2;
        // non-STL fullDeck[iCurrentCard++] = c3;
        // non-STL fullDeck[iCurrentCard++] = c4;
        fullDeck.push_back( c1 );
        fullDeck.push_back( c2 );
        fullDeck.push_back( c3 );
        fullDeck.push_back( c4 );
    }
}

void WarDeck::shuffleDeck()
{
    typedef CardVector::iterator CardVectorIt;
    CardVectorIt Start, End;
    Start = fullDeck.begin();
    End = fullDeck.end();
    random_shuffle( Start, End );
}

aSingleCard WarDeck::drawCardFromDeck()
{
    /* if( ! isDeckEmpty() )
        return fullDeck[--iCurrentCard];
    else {
        aSingleCard defaultCard( spade, 1 );
        return defaultCard;
    }
    */
    aSingleCard tempCard;
    if( !fullDeck.empty() ) {
        tempCard = fullDeck.front();
        fullDeck.erase(fullDeck.begin());
    }
    else {
        aSingleCard defaultCard( spade, 1 );
        tempCard = defaultCard;
    }
    return tempCard;
}
```

```

bool WarDeck::isDeckEmpty()
{
    // non-STL return iCurrentCard <= 0;
    return fullDeck.empty();
}
//*****
// class Opponent with defined methods
class Opponent {
public:
    Opponent( WarDeck& );
    aSingleCard drawCardFromDeck();
    void addUpPoints( int );
    int what_sTheScore();
    void putBackCard( WarDeck& );
protected:
    #define CARDS_IN_HAND 3
    // non-STL aSingleCard currentHand[CARDS_IN_HAND];
    CardVector currentHand;
    int currentScore;
    int cardBeingPlayed;
};
Opponent::Opponent( WarDeck & aDeck )
{
    currentScore = 0;
    for( int i = 0; i < CARDS_IN_HAND; i++ )
        // non-STL currentHand[i] = aDeck.drawCardFromDeck();
        currentHand.push_back(aDeck.drawCardFromDeck());
    cardBeingPlayed = 0;
}
aSingleCard Opponent::drawCardFromDeck()
{
    cardBeingPlayed = rand() % 3;
    return currentHand[cardBeingPlayed];
}
void Opponent::addUpPoints( int howMany )
{
    currentScore += howMany;
}
int Opponent::what_sTheScore()
{
    return currentScore;
}
void Opponent::putBackCard(WarDeck& aDeck)

```



```
{
    currentHand[cardBeingPlayed] = aDeck.drawCardFromDeck();
}
//*****
// main() function
void main( void )
{
    WarDeck actualDeck;
    actualDeck.shuffleDeck();
    srand( (unsigned int)time( NULL ) );
    Opponent opponent1(actualDeck);
    Opponent opponent2(actualDeck);
    while( !actualDeck.isDeckEmpty() ) {
        aSingleCard card1 = opponent1.drawCardFromDeck();
        cout << "Opponent 1 plays " << card1 << endl;
        aSingleCard card2 = opponent2.drawCardFromDeck();
        cout << "Opponent 2 plays " << card2 << endl;

        if( card1.oneCard.first == card2.oneCard.first ) {
            opponent1.addUpPoints(1);
            opponent2.addUpPoints(1);
            cout << "Players tie\n" << endl;
        }
        else if( card1.oneCard.first > card2.oneCard.first ) {
            opponent1.addUpPoints(2);
            cout << "Opponent 1 wins round\n";
        }
        else {
            opponent2.addUpPoints(2);
            cout << "Opponent 2 wins round\n";
        }
        opponent1.putBackCard(actualDeck);
        opponent2.putBackCard(actualDeck);
        cout << "\n\nPress ENTER to continue." << endl;
        cin.get();
    }
    cout << "Opponent 1 what_sTheScore " << opponent1.what_sTheScore() << endl;
    cout << "Opponent 2 what_sTheScore " << opponent2.what_sTheScore() << endl;

}
//*****
// class aSingleCard friend overloaded insertion operator
ostream& operator <<( ostream& out, aSingleCard& aCard )
```

```

{
switch( aCard.oneCard.first ) {
    case 1: out << "Ace" ; break;
    case 11: out << "Jack" ; break;
    case 12: out << "Queen"; break;
    case 13: out << "King" ; break;
    default:
        out << aCard.oneCard.first; break;
}
switch( aCard.oneCard.second ) {
    case diamond: out << " of Diamonds"; break;
    case spade : out << " of Spades "; break;
    case heart : out << " of Hearts "; break;
    case club : out << " of Clubs "; break;
}
return out;
};
    
```

13.4 小结

从本章的例子中可以看到，调试 STL 算法不需要特殊的 Visual C++ Debugger 技巧，如 F10(跳过)和 F11(跳入)这样的标准 Debugger 命令几乎是定位一个出错的 STL 语句所需要的全部命令。然而，要修复错误，却需要熟悉 STL 组件的组合关系。与 C 和 C++编译器本身那样被设计用来编写多种用途的算法不同，STL 使用标准的 C/C++产生了 STL 特定的语法和一套相互作用关系。 ■

第五部分

DEBUGGING

DEBUGGING

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

DEBUGGING

特殊的调试问题

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

- 第 14 章 使用 DLL 工作
- 第 15 章 使用 ActiveX 控件工作
- 第 16 章 调试 COM、ATL 和 DHTML
- 第 17 章 STL 和 MFC 编程



DEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGINGDEBUGGING

第 14 章

使用 DLL 工作





本章将学习如何创建和调试一个简单的动态链接库(Dynamic Link Library), 即 DLL。动态链接库和其他 Visual C++ 库一样, 使程序员能够轻松地分类新的函数和其他资源。DLL 与其他库有所不同, 因为其在运行时而不是在编译/链接时与应用程序链接的。这一过程可以称为动态链接, 与静态链接不同。静态链接发生在编译/链接期间, 将 C++ 运行时的库与应用程序链接。在多任务环境下, DLL 还具有共享函数和资源的优点。

当两个应用程序使用一个在编译期间链接的库时, 库的代码被每一个应用程序捆绑和拥有。当应用程序动态链接时, 系统为所有应用程序产生库的一个复制。然后在运行时, 可以用入口库来定位所需函数和资源。当如此处理时, 意味着所有应用程序共享同样的 DLL 函数。

DLL 可以分为明显的两类: 用 C 或 C++(不带对象)编写的常规的基于 API 的 DLL 和基于 MFC 对象的 DLL。基于 API 的 DLL 具有可方便地从一个编译器移到另一个编译器的优点; 基于 MFC 的 DLL 被限制于使用许可版本的 MFC 的编译器。本章开发的代码将演示使用 MFC 库开发一个简单的 DLL 的过程。

为 DLL 开发的源代码与本书中为其他工程开发的源代码类似, 开发一个新的 DLL 的真正问题在于调试错误的代码, 因为这时问题变得复杂了。首先, 开发了 DLL, 就必须编写一个调用该 DLL 的应用程序。如果每部分都能正常工作, 那么就没什么问题。但是如果出现了问题而需要调试, 那么将不得不考虑既要调试 DLL 又要调试调用应用程序。再加上很可能要远程调试, 情况就比第 8 章中描述的一个简单工程远程调试进行复杂多了。

我们将假定读者有一些开发 DLL 的经验, 然而本章仍将循序渐进地创建一个简单的 DLL 和使用这一 DLL 的应用程序。本章的中心是说明如何建立远程调试, 以便于正确地调试该 DLL 和使用该 DLL 的应用程序。如果需要学习更多有关开发 DLL 的内容, 可以找到这方面的书。

14.1 创建一个基于 MFC 的动态链接库

基于 MFC 的 DLL 可以以类似于其他 MFC Windows 工程的方式产生和建立。为创建本章的 DLL, 使用 AppWizard 生成需要的所有头文件、资源文件和源代码文件。工程将命名为 Framer。可以按照以下步骤创建工程 Framer:

1. 使用 Visual C++ File | New 菜单选项打开 New 对话框, 如图 14-1 所示。
2. 为新的 MFC AppWizard(dll)工程命名为 Framer。
3. 单击 OK 启动 MFC AppWizard(dll)。
4. 对于 DLL AppWizard 的 Step 1, 确保选择了 Regular DLL using shared MFC DLL 选项, 如图 14-2 所示。
5. 剩下的步骤使用 AppWizard 的缺省设定。
6. 单击 Finish, 复查图 14-3 中各项, 然后单击 OK 生成 Framer 工程的基本代码。

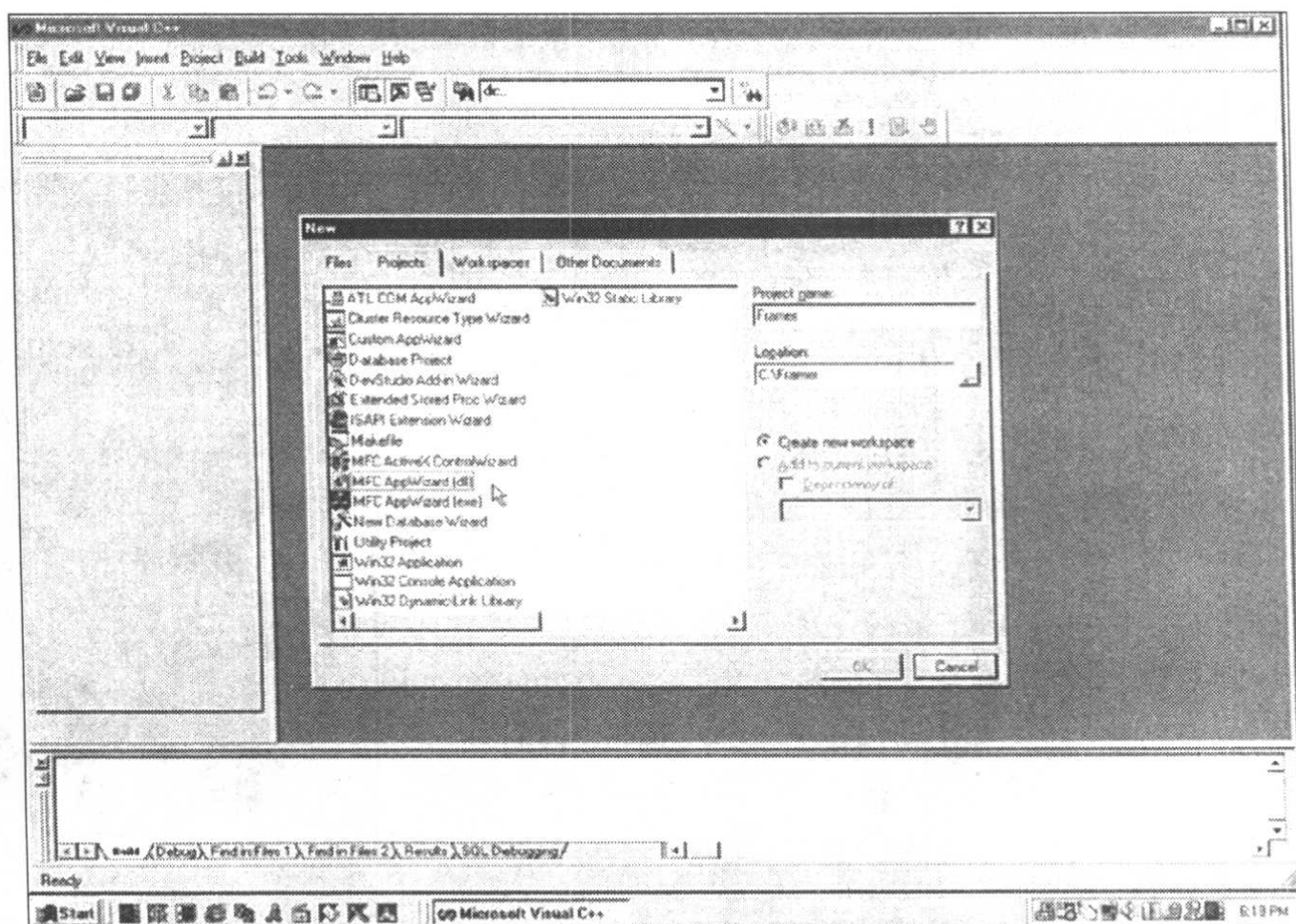


图 14-1 New 对话框允许创建一个 MFC AppWizard(dll)工程

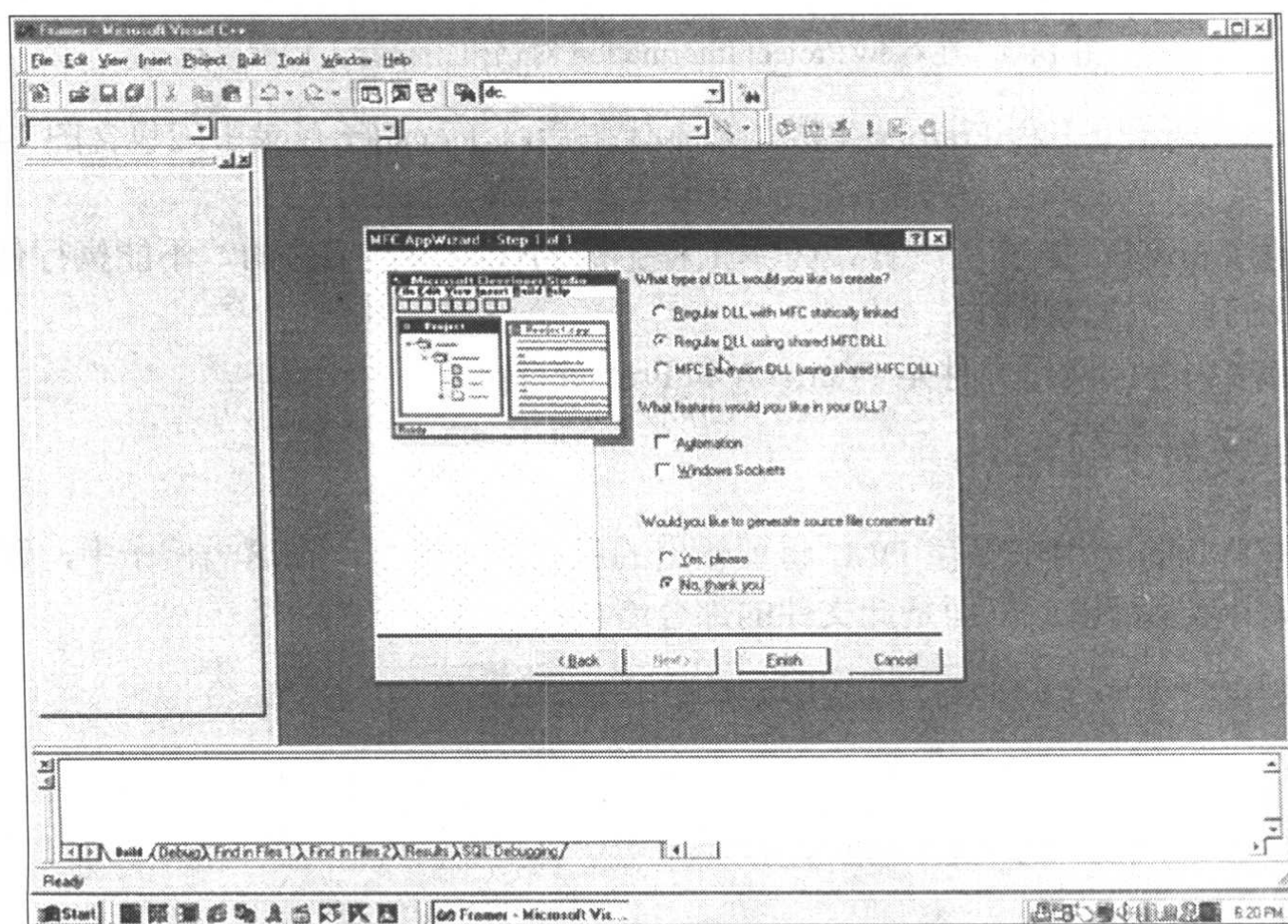


图 14-2 使用 AppWizard 的第一步选择共享 MFC DLL 选项

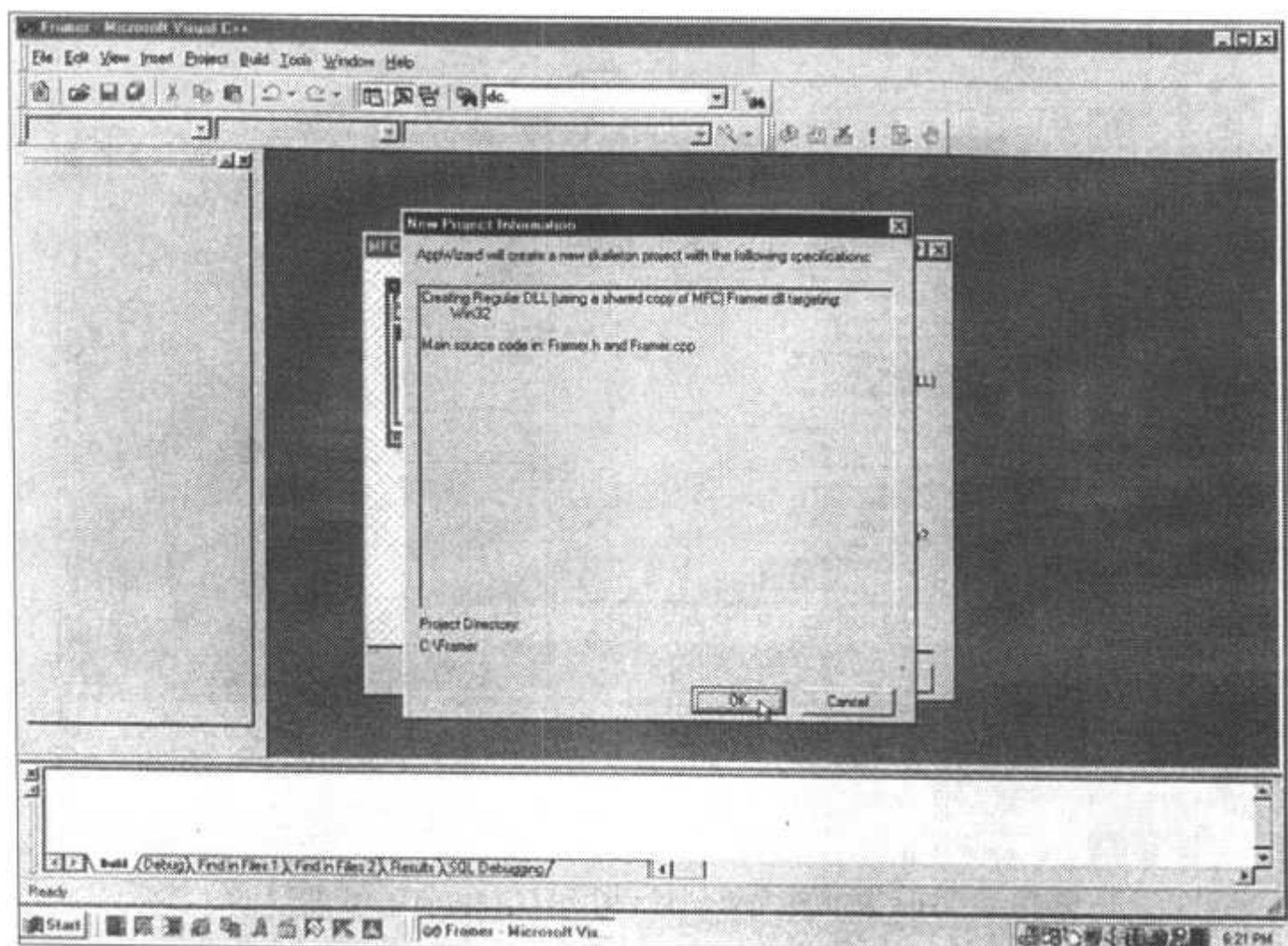


图 14-3 在 New Project Information 对话框中查看工程的信息

当 AppWizard 为工程 Framer 生成了基本代码后，工程的子目录中应包含图 14-4 中所示的文件。

和其他 AppWizard 模板一样，为本工程生成的代码为框架式的，不能执行任何操作，直到添加我们自己的代码。

我们最感兴趣的两个文件是 Framer.h 和 Framer.cpp。

14.1.1 头文件 Framer.h

头文件 Framer.h 用来保存 DLL 将要输出的所有函数原型。在这一例子中，我们添加的函数原型以黑体字示出。下面是此文件的部分清单：

```
// Framer.h : main header file for the FRAMER DLL
//
.
.
.
.
.
#if _MSC_VER > 1000
```

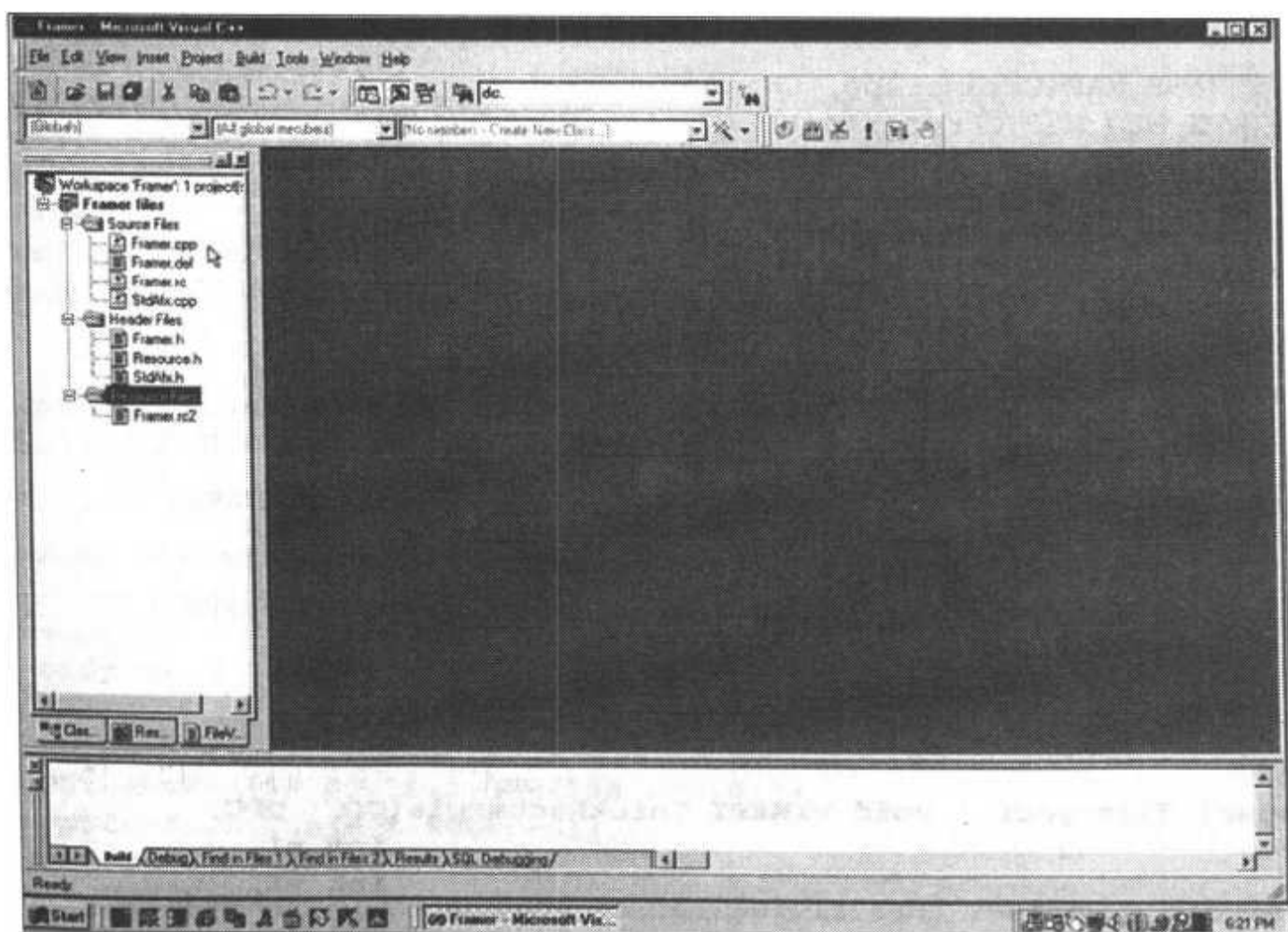


图 14-4 左边的窗格显示 AppWizard 为 Framer DLL 工程生成的文件

```
#pragma once
#endif // _MSC_VER > 1000
#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif
#include "resource.h"          // main symbols
__declspec( dllexport ) void WINAPI ThickRectangle(CDC* pDC,
                                                    int x1,
                                                    int y1,
                                                    int x2,
                                                    int y2,
                                                    int t);
__declspec( dllexport ) void WINAPI ThickEllipse(CDC* pDC,
                                                  int x1,
                                                  int y1,
                                                  int x2,
                                                  int y2,
                                                  int t);
__declspec( dllexport ) void WINAPI ThickPixel(CDC* pDC,
```



```
int x1,  
int y1);  
////////////////////////////////////  
// CFramerApp  
// See Framer.cpp for the implementation of this class  
//  
.  
.  
.
```

为简化和标准化 Microsoft 特定的对 C++ 语言的扩充部分, Microsoft 使用了一种扩展属性的语法, 如 **_declspec**。这一关键字表明, 这种类型的实例将以 Microsoft 特定的存储类属性保存。

使用关键字 **dllexport** 后, 不再需要曾经在模块定义文件中出现的旧式 **EXPORT** 语句。

14.1.2 源代码文件 Framer.cpp

现在 DLL 模板源代码中可以添加用户开发的特定代码。下面的清单以黑体字显示了添加到源文件 Framer.cpp 的代码:

```
// Framer.cpp : Defines the initialization routines for the DLL.  
//  
#include "stdafx.h"  
#include "Framer.h"  
#ifdef _DEBUG  
#define new DEBUG_NEW  
#undef THIS_FILE  
static char THIS_FILE[] = __FILE__;  
#endif  
////////////////////////////////////  
// CFramerApp  
BEGIN_MESSAGE_MAP(CFramerApp, CWinApp)  
    //{{AFX_MSG_MAP(CFramerApp)  
        // NOTE - the ClassWizard will add and remove mapping macros here.  
        // DO NOT EDIT what you see in these blocks of generated code!  
    //}}AFX_MSG_MAP  
END_MESSAGE_MAP()  
////////////////////////////////////  
// CFramerApp construction  
CFramerApp::CFramerApp()  
{  
}  
////////////////////////////////////  
// The one and only CFramerApp object
```

```

CFramerApp theApp;
__declspec( dllexport ) void WINAPI ThickRectangle(CDC* pDC,
                                                    int x1,
                                                    int y1,
                                                    int x2,
                                                    int y2,
                                                    int t)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    CBrush newbrush;
    CBrush* oldbrush;
    pDC->Rectangle(x1,y1,x2,y2);
    pDC->Rectangle(x1+t,y1+t,x2-t,y2-t);
    newbrush.CreateSolidBrush(RGB(255,255,0));
    oldbrush=pDC->SelectObject(&newbrush);
    pDC->FloodFill(x1+(t/2),y1+(t/2),RGB(0,0,0));
    pDC->SelectObject(oldbrush);
    newbrush.DeleteObject();
}

__declspec( dllexport ) void WINAPI ThickEllipse(CDC* pDC,
                                                  int x1,
                                                  int y1,
                                                  int x2,
                                                  int y2,
                                                  int t)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    CBrush newbrush;
    CBrush* oldbrush;
    pDC->Ellipse(x1,y1,x2,y2);
    pDC->Ellipse(x1+t,y1+t,x2-t,y2-t);
    newbrush.CreateSolidBrush(RGB(255,255,0));
    oldbrush=pDC->SelectObject(&newbrush);
    pDC->FloodFill(x1+(t/2),y1 + ((y2 - y1)/2),RGB(0,0,0));
    pDC->SelectObject(oldbrush);
    newbrush.DeleteObject();
}

__declspec( dllexport ) void WINAPI ThickPixel(CDC* pDC,
                                                int x1,
                                                int y1)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
}

```



```
CPen newpen;  
CPen* oldpen;  
pDC->SetPixel(x1,y1,0L);  
newpen.CreatePen(PS_SOLID,2,RGB(255,255,0));  
oldpen=pDC->SelectObject(&newpen);  
pDC->MoveTo(x1-5,y1);  
pDC->LineTo(x1-1,y1);  
pDC->MoveTo(x1+1,y1);  
pDC->LineTo(x1+5,y1);  
pDC->MoveTo(x1,y1-5);  
pDC->LineTo(x1,y1-1);  
pDC->MoveTo(x1,y1+1);  
pDC->LineTo(x1,y1+5);  
pDC->SelectObject(oldpen);  
newpen.DeleteObject();  
}
```

设计提示

Microsoft 为动态链接至 MFC DLL 的所有 DLL 做了以下建议。任何从这些 DLL 中输出的、进入 MFC 调用的函数，必须在函数的最开始处加上宏 `AFX_MANAGE_STATE`。例如：

```
// extern "C" BOOL PASCAL EXPORT ExportFunction()  
// {  
//     AFX_MANAGE_STATE(AfxGetStaticModuleState());  
//     // normal function body here  
// }  
//
```

在任何进入 MFC 的调用之前，这一宏必须在每个函数中出现。换句话说，在函数内它必须是第一条语句，甚至在任何对象声明之前。这是因为其构造函数可能产生进入 MFC DLL 的调用。Microsoft 提供的 MFC Technical Notes 33 和 58 可以作为附加信息。

这一 DLL 提供了三个绘画程序：`ThickRectangle()`、`ThickEllipse()`和 `ThickPixel()`。当其被主应用程序调用并被传给正确的参数时，前两个函数将分别画出宽边的矩形和椭圆，宽边被填充以预先定义的颜色。函数 `ThickPixel()`将在像素位置用单一的颜色画一个十字线。

当 DLL 被捆绑在主应用程序上时，DLL 函数可以被传递给适当的实际参数，用函数名调用。

14.1.3 建立 Framer.dll

DLL 可以通过从 Visual C++ Build 菜单中选择合适的选项建立。当建立过程完成后，debug 子目录将含有几个重要文件。

文件 Framer.dll 为动态链接库，而 Framer.lib 为相关联的库。两个文件都必须放在特定的位置。

1. 将 Framer.dll 复制到包含系统 DLL 的 Windows 子目录下，对于 Windows 98 通常为 C:\Windows\System。

2. 将 Framer.lib 复制到使用该 DLL 的应用程序的 debug 子目录下，本例的子目录命名为 C:\DLLDemo\debug。

为测试该 DLL，我们不得不建立一个标准的 MFC 应用程序调用它。

14.2 创建使用 DLL 的主应用程序

现在我们需要创建一个应用程序，使用 Framer.dll 动态链接库。这一应用程序命名为 DLLDemo，将调用 DLL 的每个函数若干次。

按照以下步骤，使用 AppWizard 生成 DLLDemo 工程的模板代码：

1. 选择 Visual C++ File | New 菜单选项打开 New 对话框，如图 14-5 所示。

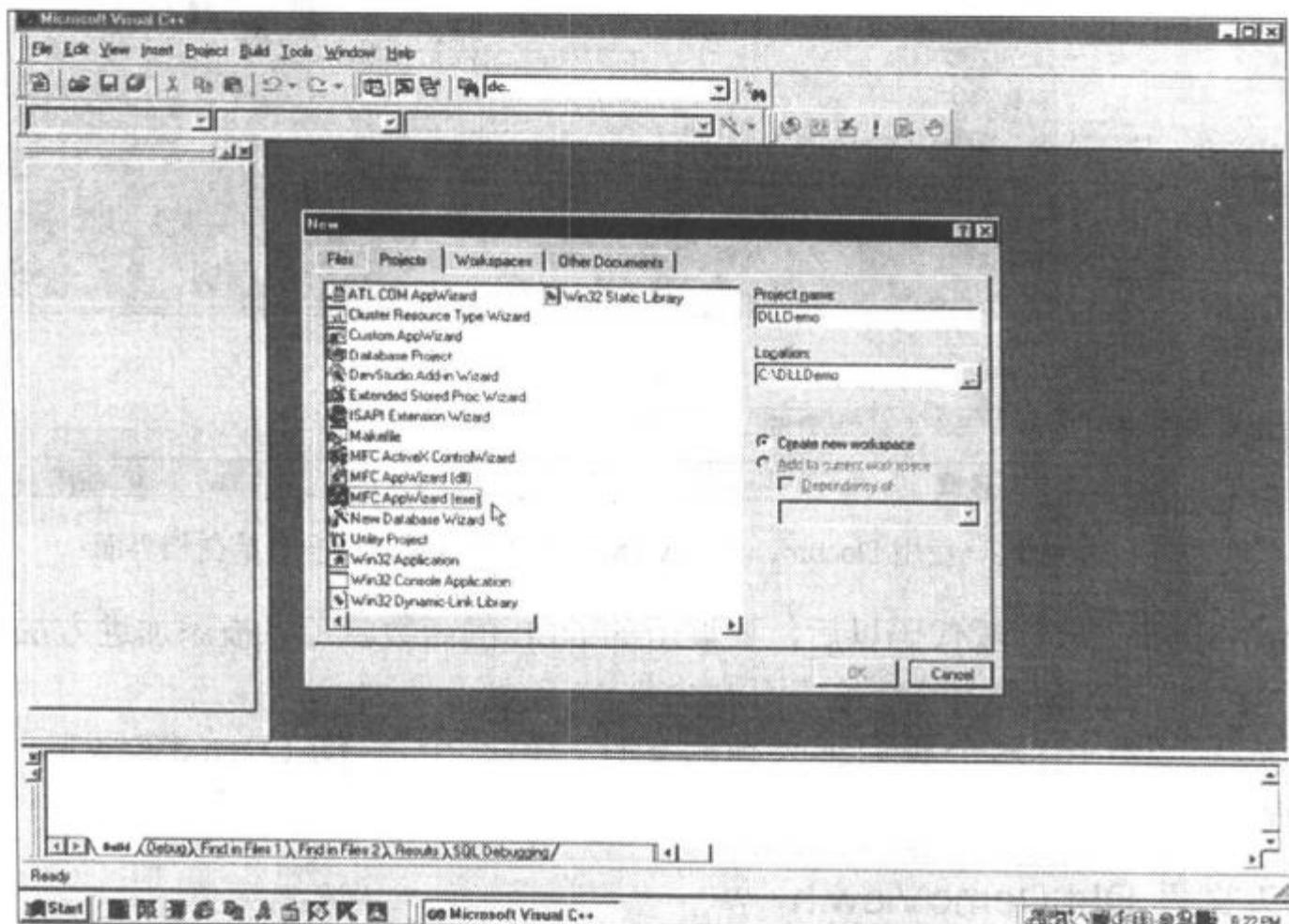


图 14-5 使用 New 对话框创建一个名为 DLLDemo 的 MFC AppWizard 应用程序

2. 为工程命名为 DLLDemo，单击 OK，启动 MFC AppWizard。



3. 按照常规的创建工程的步骤，生成一个带有单文档接口，使用 Document/View 体系结构支持的应用程序，如图 14-6 所示。
4. 其余步骤使用 wizard 的缺省设定，建立一个普通的工程。
5. 查看并接受复查列表中显示的类，单击 Finish 按钮，生成工程文件。

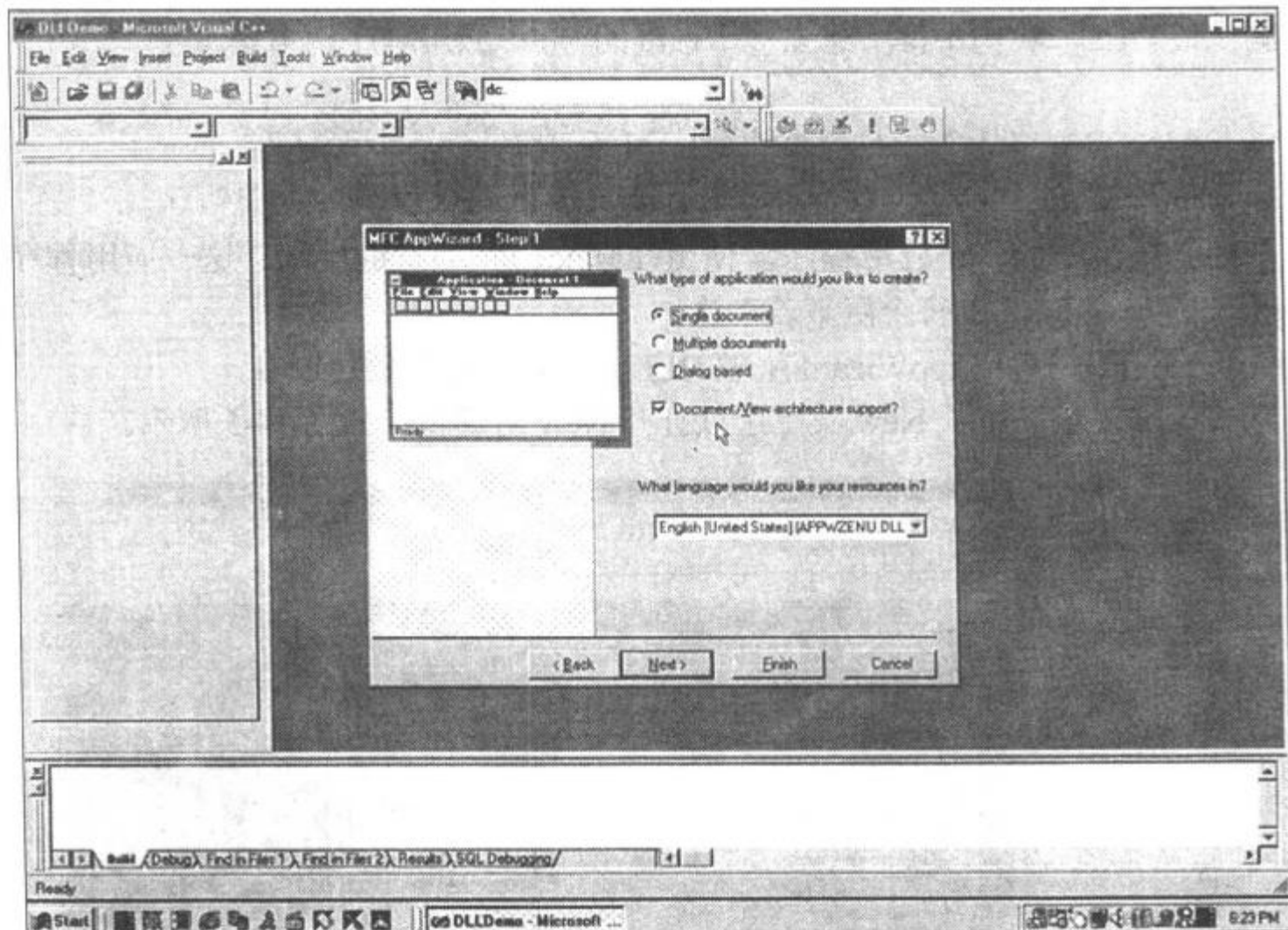


图 14-6 为这一使用 Document/View 体系结构的工程选择一个单文档界面

在 AppWizard 生成模板代码以后，要调用的 DLL 的函数就可以被添加进方法 OnDraw() 中了。

再一次，工程中有两个我们感兴趣的文件，第一个是 DLLDemoView.h，第二个是 DLLDemoView.cpp。

14.2.1 头文件 DLLDemoView.h

头文件 DLLDemoView.h 用来保存那些我们希望从 Framer.dll 中输出的函数原型，下面的部分清单以黑体字显示了这些函数原型：

```
// DLLDemoView.h : interface of the CDLLDemoView class
```

```
//
//
//
.
.
.
extern void WINAPI ThickRectangle(CDC* pDC,int x1,int y1,int x2,int y2,int
t);
extern void WINAPI ThickEllipse(CDC* pDC,int x1,int y1,int x2,int y2,int t);
extern void WINAPI ThickPixel(CDC* pDC,int x1,int y1);
class CDLLDemoView : public CView
{
.
.
.
}
```

关键字 **extern** 提示编译器，这些函数相对当前函数体是外部的。在建立过程中，链接程序将寻找这些函数中的每一个。如果链接程序在合适的库中找不到任何一个函数，都将发布一个简短的错误信息。

设计提示

当编译器遇到一个不是普通 GDI 绘画函数的函数时，比如 `ThickRectangle()`，它要发布一个错误信息。通过使用关键字 **extern**，向编译器承诺有关这一函数的信息将在链接时提供（通常在一个 DLL 中或其他库中）。如果链接程序不能发现承诺的这一函数信息，它将为之发布一个错误信息。

14.2.2 源代码文件 DLLDemoView.cpp

以下是源代码文件 `DLLDemoView.cpp` 的一个清单，负责调用 DLL 函数的代码以黑体字示出：

```
// DLLDemoView.cpp : implementation of the CDLLDemoView class
//
#include "stdafx.h"
#include "DLLDemo.h"
#include "DLLDemoDoc.h"
#include "DLLDemoView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```



```
////////////////////////////////////
// CDLLDemoView
IMPLEMENT_DYNCREATE(CDLLDemoView, CView)
BEGIN_MESSAGE_MAP(CDLLDemoView, CView)
    //{AFX_MSG_MAP(CDLLDemoView)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
// CDLLDemoView construction/destruction
CDLLDemoView::CDLLDemoView()
{
}
CDLLDemoView::~CDLLDemoView()
{
}
BOOL CDLLDemoView::PreCreateWindow(CREATESTRUCT& cs)
{
    return CView::PreCreateWindow(cs);
}
////////////////////////////////////
// CDLLDemoView drawing
void CDLLDemoView::OnDraw(CDC* pDC)
{
    CDLLDemoDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // Call ThickPixel() several times
    for (int i=50;i<900;i+=100)
        ThickPixel(pDC,i,75);
    // Call ThickRectangle() several times
    ThickRectangle(pDC,50,300,75,350,20);
    ThickRectangle(pDC,150,350,250,450,25);
    ThickRectangle(pDC,400,200,700,600,25);
    // Call ThickEllipse() several times
    ThickEllipse(pDC,50,100,75,150,10);
    ThickEllipse(pDC,150,150,250,250,15);
    ThickEllipse(pDC,450,250,650,550,10);
}
////////////////////////////////////
// CDLLDemoView diagnostics
#ifdef _DEBUG
void CDLLDemoView::AssertValid() const
{
    CView::AssertValid();
}
void CDLLDemoView::Dump(CDumpContext& dc) const
```

```
{
    CView::Dump(dc);
}
CDLLDemoDoc* CDLLDemoView::GetDocument() // non-debug inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CDLLDemoDoc)));
    return (CDLLDemoDoc*)m_pDocument;
}
#endif // _DEBUG
////////////////////////////////////
// CDLLDemoView message handlers
```

在建立这一工程之前，还有关键的一步要执行。必须标识 `Framer.lib` 以使链接程序能够分解外部函数调用。这需要使用编译器的 **Project | Settings** 菜单选项打开 **Project Settings** 对话框，图 14-7 显示了这一对话框，其中 **Link** 页被选中。

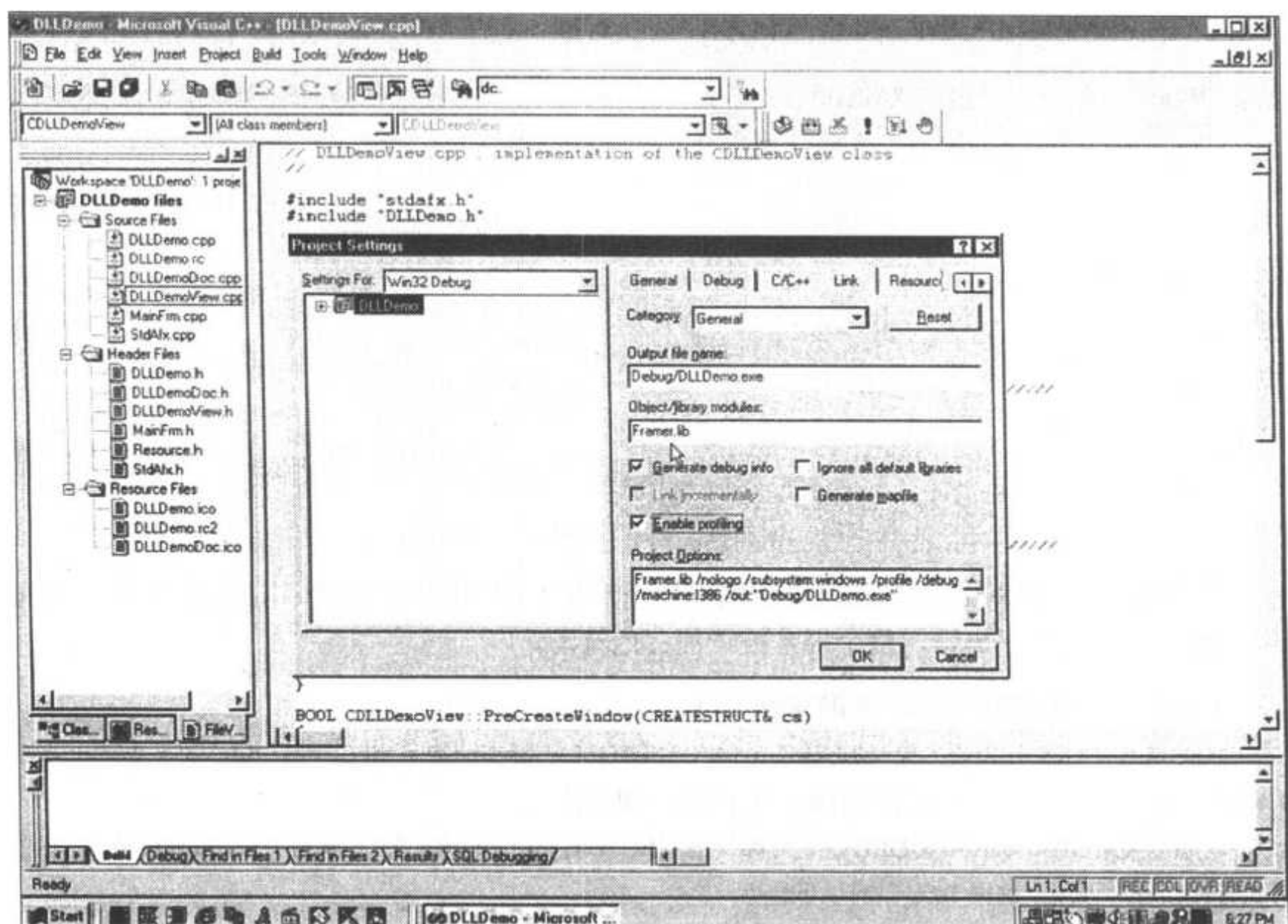


图 14-7 使用 **Project Settings** 对话框的 **Link** 标签下的 **Object/library module** 编辑框标识



现在可以从编译器的 Build 菜单中选择正确的命令建立应用程序。
运行该应用程序，应当看到类似于图 14-8 的屏幕。

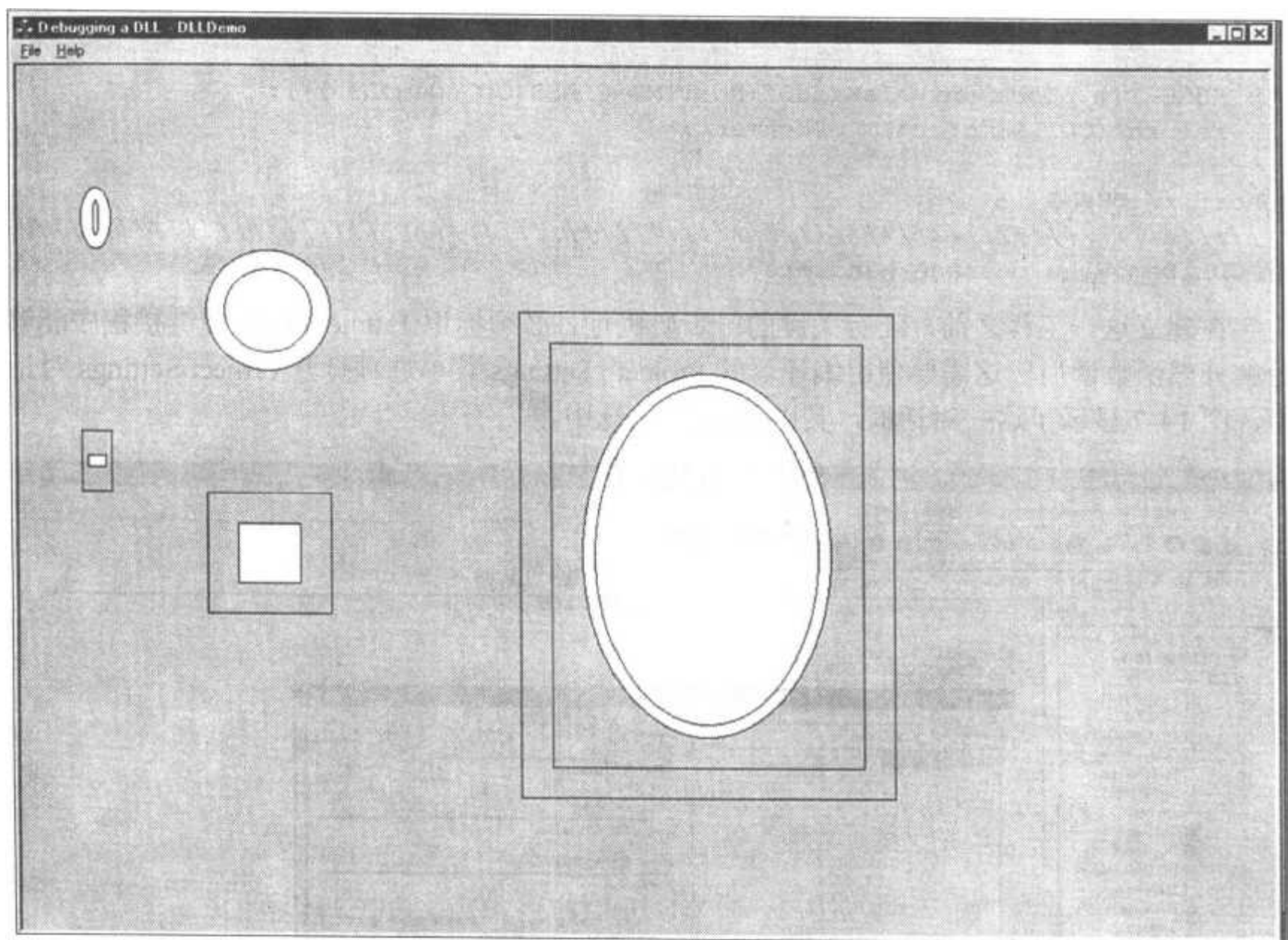


图 14-8 应用程序 DLLDemo 有一个问题

应用程序 DLLDemo 应当在屏幕上画出几个形状不同的图形，每个图形的宽边以黄色填充。图 14-8 表明程序出错，整个屏幕被一片黄色淹没。

24x7 动态链接至 MFC 库的常规 DLL

Microsoft 将 DLL 分为两大类：动态链接至 MFC 库的 DLL 和静态链接至 MFC 库的 DLL。以下是 Microsoft 为动态链接至 MFC 库的 DLL 提供的信息，其中做了一些修改。

动态链接至 MFC 库的常规 DLL 是在内部使用 MFC 库的 DLL，输出的 DLL 函数可以被 MFC 或非 MFC 可执行应用程序调用，这种 DLL 具有以下特性：

- DLL 是动态地链接至 MFC DLL 的。
- 使用 DLL 的主应用程序可以用任何支持使用 DLL 的语言编写的，并且不必是基于 MFC 的应用

程序。

- 链接至这种 DLL 的 MFC 输入库与用作扩充 DLL 的库或使用 MFC DLL 的应用程序是一样的。

为了动态链接一个常规的 DLL，必须满足以下要求：

1. 常规 DLL 必须有一个 CWinApp 的派生类和应用程序类的单一对象。DLL 的 CWinApp 对象没有主消息泵。这使其与普通应用程序的 CWinApp 对象不同。

2. DLL 是在定义了 _AFXDLL 的情况下编译的。在 _USRDLL 也定义了的情况下，这与动态链接至 MFC DLL 的可执行应用程序类似。

3. DLL 实例化为 CWinApp 派生类。

4. DLL 使用 MFC 支持的 DllMain。所有 DLL 特定的初始化代码放在成员函数 InitInstance() 中，而终止代码放在成员函数 ExitInstance() 中。

5. 在每一个由 DLL 输出的函数的开始使用宏 AFX_MANAGE_STATE，这样做是必要的，因为这种类型的 DLL 使用 MFC 的动态链接库形式。

必须用程序员自己的应用程序为 MFCx0.dll 和 Msvert.dll(或类似的文件)分配 DLL。

符号可以通过使用标准 C 接口由常规 DLL 输出。接口可以是下面的形式：

```
extern "C" _declspec( dllexport ) ExportFunctionName( );
```

常规 DLL 内的所有内存分配都应当停留在 DLL 内；该 DLL 不应传递给调用可执行程序或自其中接收任何以下内容：

动态链接至 MFC 的 DLL 使用宏 AFX_MANAGE_STATE，以正确改变 MFC 模块状态。这一代码必须被添加至所有由 DLL 输出的函数的开始处，格式如下：

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

这个宏不应当用在扩展 DLL 中，也不能与静态链接至 MFC 的常规 DLL 一同使用。

24x7 静态链接至 MFC 库的常规 DLL

静态链接至 MFC 库的 DLL 是 Microsoft 归纳的第二类 DLL，只在 Visual C++ 专业版和企业版中支持静态链接至 MFC 库的 DLL。

常规 DLL 可以静态链接至 MFC 库，在这种情况下，DLL 在内部使用 MFC。DLL 输出的函数可以由 MFC 或非 MFC 可执行应用程序调用。

静态链接的常规 DLL 具有以下特性：

- 这个 DLL 可以链接至应用程序使用的同一 MFC 静态链接库，无需一个专为 DLL 的静态链接库的分开版本。

- 可执行应用程序可以用任何支持使用 DLL 的语言编写的，并且不必是基于 MFC 的应用程序。

为将一个常规 DLL 静态链接至 MFC，必须满足以下要求：

1. DLL 必须实例化一个 CWinApp 的派生类。



2. DLL 使用 MFC 提供的 DLLMain。所有 DLL 特定的初始化代码放在成员函数 InitInstance() 中，而终止代码放在成员函数 ExitInstance() 中。这与常规 MFC 应用程序类似。

3. 术语 _USRDLL 必须在编译器命令行中定义。(注意：术语 USRDLL 已作废了，但仍须定义。)

DLL 必须有一个 CWinApp 的派生类和该应用程序类的单一对象。DLL 的对象 CWinApp 没有主消息泵，这使其与普通 MFC 应用程序的对象 CWinApp 不同。

符号可以使用标准 C 接口由常规 DLL 输出，下面是建议的格式：

```
extern "C" _declspec( dllexport ) ExportFunctionName( );
```

静态链接至 MFC 的 DLL 也不能动态链接至共享 MFC DLL。静态链接至 MFC 的 DLL 动态地捆绑到应用程序上，正如任何其他 DLL 一样。换句话说，应用程序链接至该 DLL，正如链接至任何其他 DLL 一样。

无需手动指定(对链接程序)将要链接的 MFC 库的版本。相反，MFC 的头文件自动确定将要链接的 MFC 库的正确版本，这是由预定义(如 _DEBUG)决定的。

24x7 调试 DLL 所用的安全技术

有许多途径调试 DLL 和相应的主应用程序。Microsoft 为以下不同情况分别概括了所需技术：使用应用程序的工程调试 DLL，使用 DLL 的工程调试 DLL，调试用外部工程创建的 DLL。以下建议适用于所有的调试过程。

- 如果 DLL 和主应用程序的源代码都存在，那么打开主应用程序的工程，从其中开始调试。如果 DLL 是动态安装的，那么它必须在 Project Settings 对话框的 Debug 标签中的 Category 处选择 Additional DLLs 类别。

- 如果只有 DLL 的源代码可得，打开建立该 DLL 的工程。使用 Project Settings 对话框中的 Debug 标签指定调用该 DLL 的主应用程序。

- 如果只有 DLL 和源代码，而没有工程，使用 File | Open 菜单选项选择将要调试的 DLL 文件。调试信息应当包含在 DLL 文件或相应的 PDB 文件中。选择 Build | Start Debug，然后选择 Go，开始调试过程。

按照以下步骤使用应用程序的工程文件调试 DLL：

1. 选择 Project | Settings 选项。
2. Project Settings 对话框打开，选择 Debug 标签。
3. 选择 Category 下拉列表中的 General。
4. 使用 Program Argument 文本框输入主应用程序需要的任何命令行参数。
5. 利用 Category 下拉列表选择任何另外的 DLL(Additional DLLs)。
6. 使用 Local Name 栏输入要调试的 DLL 的名称。
7. 针对远程调试，将出现 Remote Name 栏。为映射至本地模块名称的远程模块输入完整的路径。
8. 使用 Preload 栏(如果存在的话)复选 load the module before debugging begins if desired。

9. 单击 OK 将这些信息保存在工程中。
10. 选择 Build | Start Debug 菜单项，然后选择 Go，启动 Debugger。

在 DLL 和主应用程序中都可以设置断点。

按照以下步骤使用 DLL 的工程文件调试 DLL：

1. 选择 Project | Settings 选项。
2. Project Settings 对话框打开。选择 Debug 标签。
3. 选择 Category 下拉列表中的 General。
4. 使用 Executable For Debug Session 文本框输入调用该 DLL 的可执行文件的名称。
5. 利用 Category 下拉列表选择任何另外的 DLL (Additional DLLs)。
6. 使用 Local Name 栏输入要调试的 DLL 的名称。
7. 单击 OK 将这些信息保存在工程中。
8. 选择 Build | Start Debug 菜单项，然后选择 Go，启动 Debugger。

在 DLL 和主应用程序中仍然都可以设置断点。

按照以下步骤调试用外部工程创建的 DLL：

1. 选择 Project | Settings 选项。
2. Project Settings 对话框打开，选择 Debug 标签。
3. 选择 Category 下拉列表中的 General。
4. 使用 Executable For Debug Session 文本框输入由外部工程创建的 DLL 的名称。
5. 单击 OK 将这些信息保存在工程中。
6. 建立一个含有符号调试信息的该 DLL 的版本。
7. 选择 Build | Start Debug 菜单项，然后选择 Go，启动 Debugger。

在 DLL 和主应用程序中仍然都可以设置断点。这可以在 DLL 的最终建立产生之前完成。

14.3 更加仔细地查看

检查图 14-8 时，无疑会注意到窗口中输出结果的一个问题。每个图形的宽边没有填充，相反，屏幕被填充了。这时我们知道 DLL 是部分工作的，因为图形被画出。我们所不知道的是哪个例行程序出错或有多少个例行程序出错。

14.3.1 远程调试

在下面几部分，将学习如何为远程调试设置两台计算机。除了原来在第 8 章中给出的信息以外，又加上一项任务，现在既要调试主应用程序，又要调试 DLL。为保持连贯性，我们将调整第 8 章中提出的远程调试的要点，将其修改为适合 DLL 远程调试。就文件的内



容而言，关于如何为这种类型的通信设置两台计算机有许多种方法。就文件复制而言，我们的技术可能有一些大材小用，但对于在多个建筑物之间进行的来回长距离通信，可能是最容易的。

以下步骤假定 DLL(Framer)和主应用程序(DLLDemo)都已经成功生成，并且都包含必要的调试信息。

14.3.1.1 准备远程目标计算机

回想一下，远程目标计算机上必须运行一个名为 Remote Debug Monitor 的小程序。在我们的例子中，这台计算机是 Sony 的。这一应用程序和这台计算机负责与主计算机上运行的 Debugger 通信。在这一例子中，主计算机是 hp 的。软件 Remote Target Computer 控制 Debugger 中应用程序的执行。

为安装 Remote Debug Monitor，远程目标计算机(Sony)还需要其他文件。对于 Windows 98 和 Windows NT(2000)，这些文件包括：

```
MCVCMON.EXE  
MSVCRT.DLL  
TLN0T.DLL  
DM.DLL  
MSVCP60.DLL  
MSDIS110.DLL  
PSAPI.DLL (仅对 NT)
```

使用计算机的 Start 窗口中的 Find 选项在主计算机(hp)的正确子目录中查找这些文件。这些文件应当从主计算机(hp)复制到远程目标计算机(Sony)，并且保存在远程目标计算机的 Windows 子目录中。唯一的例外是文件 MSVCRT.DLL 应当复制到 Windows 98 或 Windows 2000 的 Windows\System32 子目录中。一旦完成这些，重新启动计算机。

现在，要在远程目标计算机(Sony)上运行软件 Remote Target Computer，按照以下步骤进行：

1. 在远程目标计算机(Sony)上运行应用程序 MSVCMON.EXE。
2. Visual C++ Debug Monitor 对话框出现，选择 Settings 选项。
3. 现在，出现 Win32 Network (TCP/IP) Settings 对话框。在这一对话框中输入主计算机的名字(在本例中为 hp)，(也可以使用 IP 地址)。
4. 如果口令文本框激活，则输入一个口令。这一口令在两台计算机上必须匹配；否则这个框也可以为空。
5. 单击 OK 按钮。
6. 单击 Connect 按钮。

上述步骤一旦完成，屏幕上将出现 Connecting 对话框，这时无需操作。一旦实际调试开始，这一对话框将自动消失。

一旦完成调试过程，则选择 Disconnect 按钮终止远程连接。

14.3.1.2 准备主计算机

主计算机(hp)必须准备好与远程目标计算机(Sony)通信。

在主计算机(hp)上执行以下步骤：

1. 在 Visual C++ 中，选择 Build | Debugger Remote Connection 菜单项。
2. 出现 Remote Connection 对话框。如果 Platform 下拉列表允许选择，则选择正确的平台。如果没有提供任何选项，那么缺省项被自动选择。
3. 使用 Connection 下拉列表选择 Network(TCP/IP)连接选项。
4. 选择 Settings 选项。
5. 出现 Win32 Network (TCP/IP) Settings 对话框。在此对话框中输入远程目标计算机的名字(本例中为 Sony)。也可以使用 IP 地址。如果有口令选项，则输入一个与远程目标计算机相同的口令。在远程计算机和主机中，这一框也都可以为空。
6. 单击 OK 按钮关闭 Win32 Network (TCP/IP) Settings 对话框。
7. 单击 OK 按钮关闭 Remote Connection 对话框。

上述步骤一旦完成，则设置主计算机好与远程目标计算机通信。

14.3.1.3 开始调试过程

当网络中的两台计算机做好通信准备之后，调试过程就可以开始。

错误监视

主计算机(hp)和远程计算机(Sony)都应当在正确的子目录下包含工程 Framer 和 DLLDemo 的所有文件。在主计算机(hp)中建立工程 Framer 和 DLLDemo，然后将其复制到远程计算机(Sony)的子目录中。

记住，运行 Visual C++ 的主计算机是 hp 计算机，而远程目标计算机是 Sony 计算机。建议完成以下步骤：

1. 把要调试的两个工程全部复制到两台计算机中，还建议使用同样的目录。例如，在两台计算机中，复制的文件都保存在 c:\myproject 中，其中调试可执行程序放在 c:\myproject\debug 中。记住在两台计算机中的 Framer.dll 都应当放在 C:\Windows\System 子目录中。同样，在两台计算机中的 Framer.lib 都应当放在 C:\DLLDemo\Debug 子目录中。
2. 在网络中两个目录都应当设置为共享目录。
3. 启动 Visual C++ 编译器，加载工程到主计算机(在远程目标计算机(Sony)中不需要



Visual C++编译器)。

4. 选择 Project | Settings 菜单选项，出现 Project Settings 对话框。
5. 选择 Project Settings 对话框中的 Debug 标签。
6. 从 Category 列表中选择 General，然后设置以下各项：
 - Category 列表框：输入工程所需要的任何其他 DLL，然后选择建议搜索其他 DLL 的复选框。
 - Executable for Debug Session 编辑框：按照 Debugger 在主计算机(hp)上所看到的输入可执行文件的名称和路径——在本例中是 C:\DLLDemo\debug\DLLDemo.exe。
 - Working Directory 文本框：空。
 - Program Arguments 文本框：空，除非程序需要初始参数。
 - Remote Executable Path 文本框：按照在远程目标计算机(Sony)上所看到的输入可执行文件的名称和路径——在本例中为，C:\DLLDemo\debug\DLLDemo.exe。
7. 选择 Build | Start Debug 菜单项，然后使用 Step Into(F11)选项。
8. 给系统至少一分钟接通网络，然后以正常方式开始调试。
9. 记住，每在工程中做一次改动，两台机器的所有子目录(包括 C:\Windows\System)中的文件都必须更新。

搜索任何遗漏的 DLL 及其相关的符号信息。

在下一节，将开始实际搜索 Framer.dll 中的问题，猜一猜这一问题是什么和在何处？

14.3.2 有问题的代码

为确定 DLL 中的问题，我们将使用以下这些特定的步骤，用 DLL 的工程文件调试 DLL：

1. 选择 Project | Settings 选项。
2. 出现 Project Settings 对话框，选择 Debug 标签。
3. 选择 Category 下拉列表中的 General。
4. 使用 Executable For Debug Session 文本框输入调用该 DLL 的可执行文件的名称，如图 14-9 所示。
5. 利用 Category 下拉列表选择任何另外的 DLL(Additional DLLs)。
6. 使用 Local Name 栏输入要调试的 DLL 的名称。还要为远程调试输入远程计算机中的 DLL 名称，如图 14-10 所示。
7. 单击 OK 将这些信息保存在工程中。

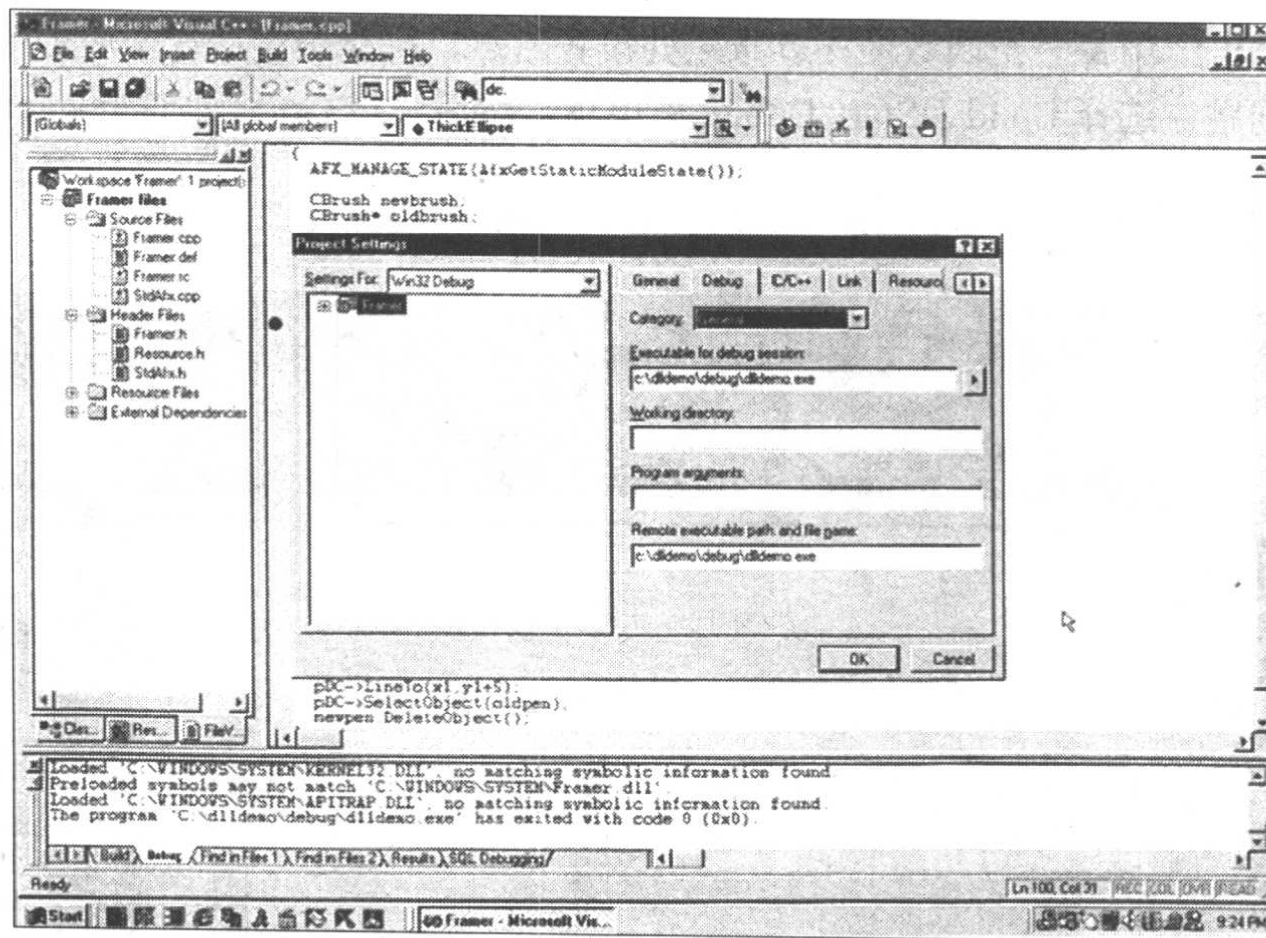


图 14-9 使用 Project Settings 对话框为调试过程设置路径

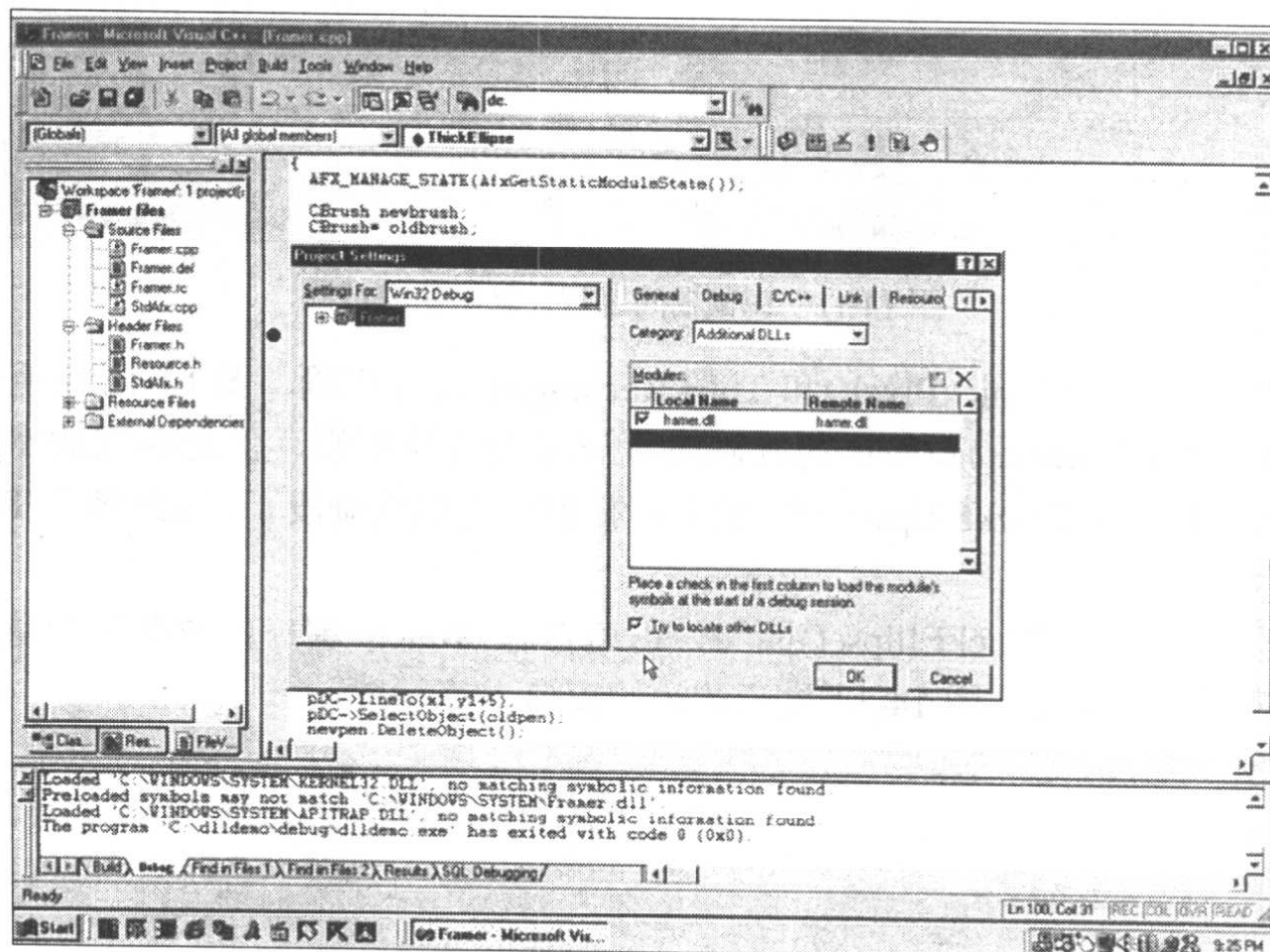


图 14-10 使用 Project Settings 对话框指定 Additional DLLs



我们在 DLL 中每一个函数的接近末尾处设置断点，以能确定这三个函数中的哪一个使填充出现了问题。选择 Build | Start Debug 菜单项，然后选择 Go 启动 Debugger 运行至第一个断点。

图 14-11 显示了运行几次 Go(F5)命令后的远程调试计算机(Sony)的屏幕。

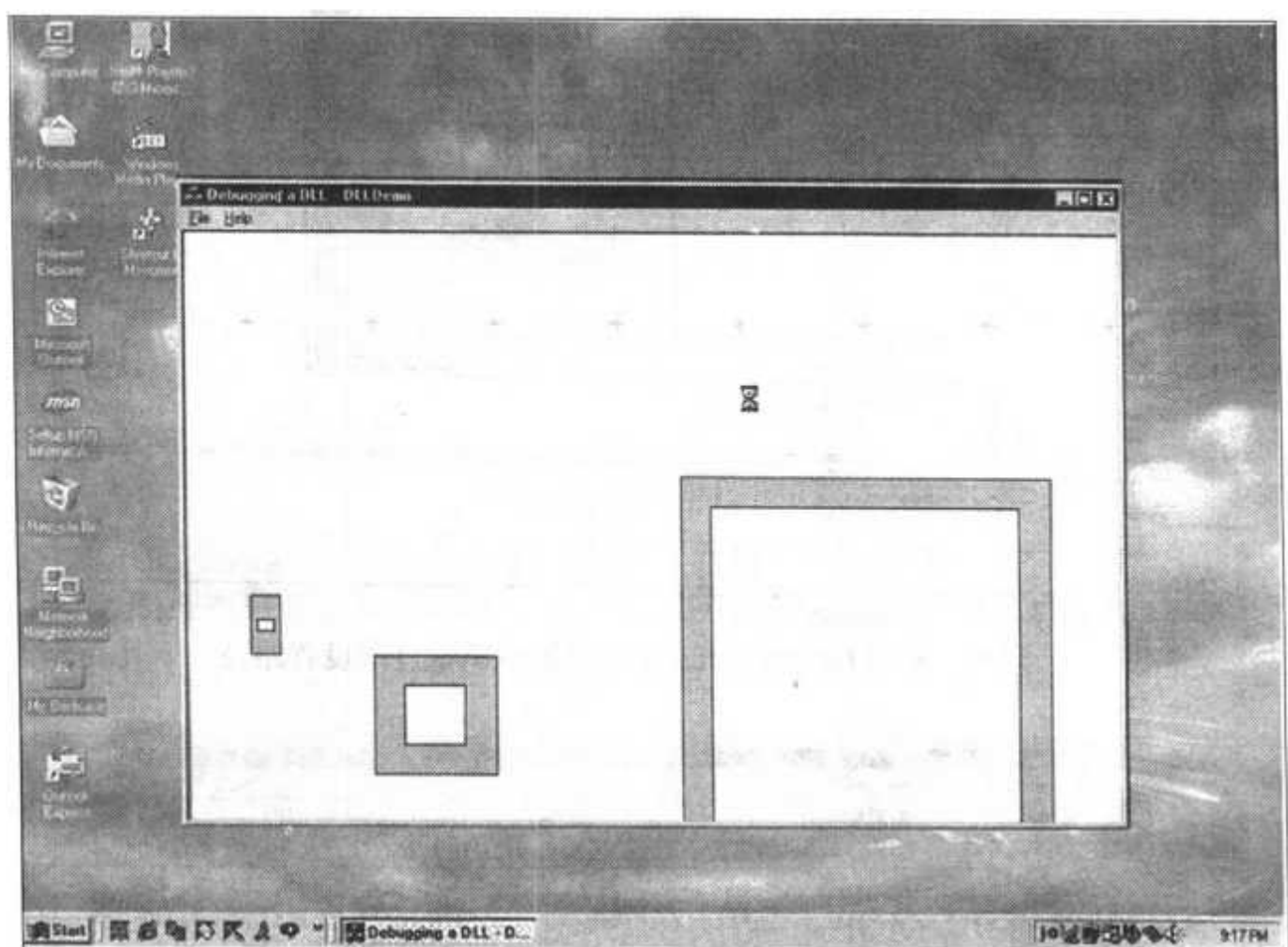


图 14-11 调试期间远程计算机的窗口

看来 DLL 中的函数 `ThickPixel()` 和 `ThickRectangle()` 运行正确，图 14-12 说明了另一点。

调用函数 `ThickEllipse()` 画最小的椭圆时，屏幕没有被淹没，但是椭圆的宽边也没有被填充。第二次调用函数 `ThickEllipse()` 时，整个屏幕被填充颜色淹没了，因此函数 `ThickEllipse()` 是问题的起因。

在第二次调用函数 `ThickEllipse()` 期间，迅速查看 Watch 窗口中放置的变量，然后简单地算一下，就揭示了问题。图 14-13 显示此时的窗口。

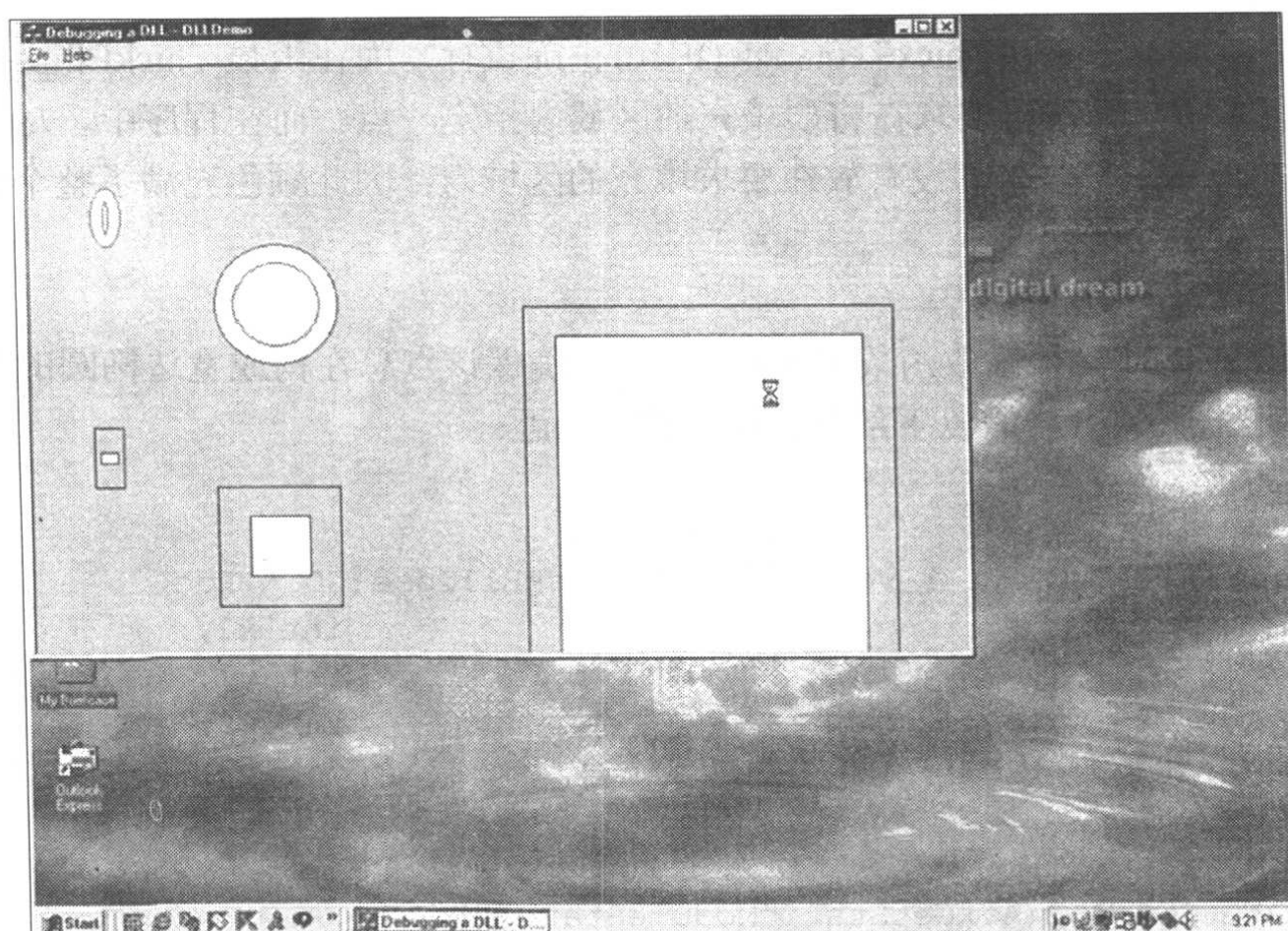


图 14-12 在调用 ThickEllipse()过程中屏幕充满了颜色

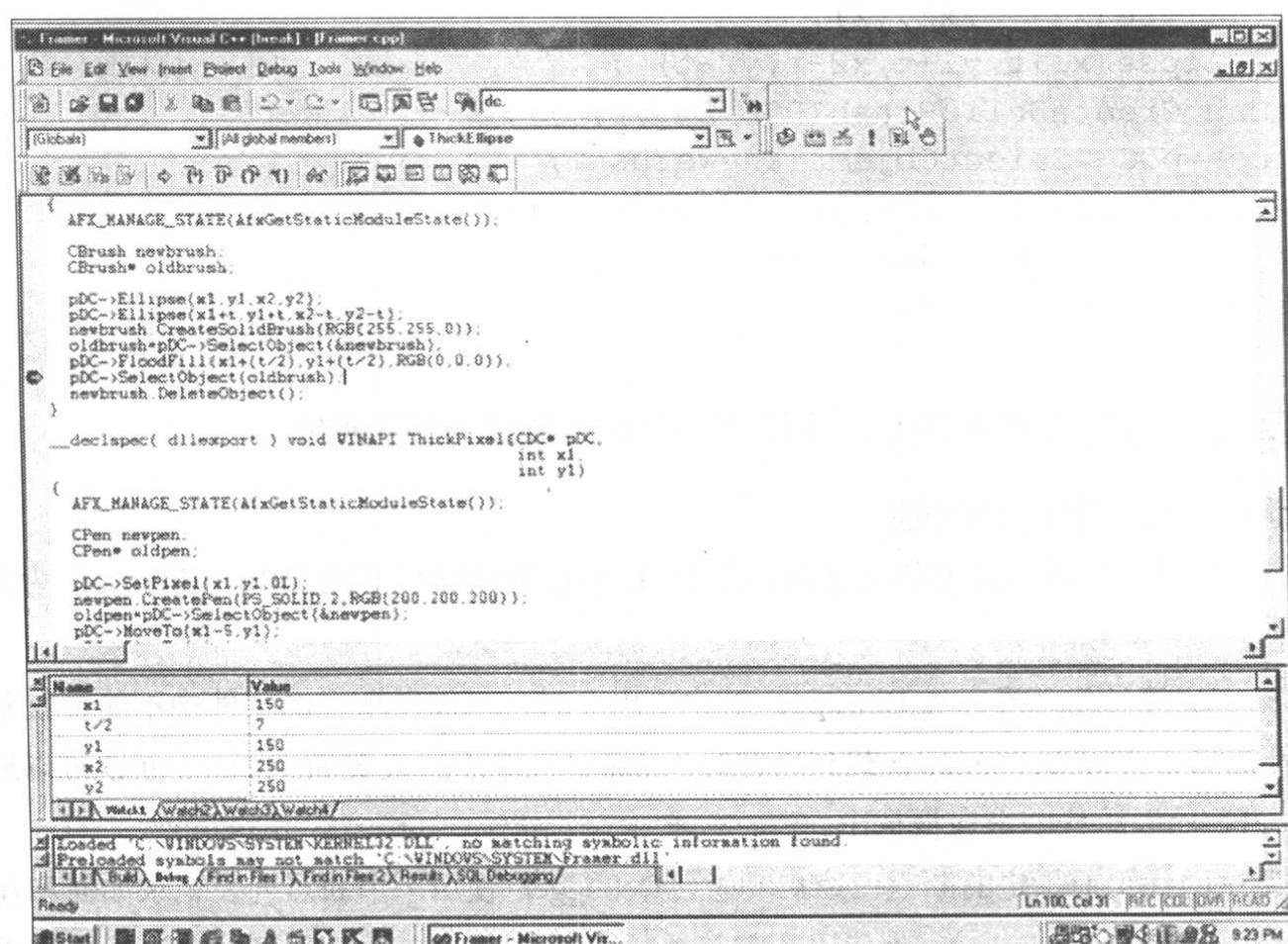


图 14-13 检查 Watch 窗口中的变量



函数 FloodFill() 在函数 ThickRectangle() 中可正确执行，而在函数 ThickEllipse() 中却不能正确执行，函数 FloodFill() 要求在将要填充的区域内指定一点。而在程序中，尽管函数将此点放在椭圆的边界矩形内，却没有放在要求填充的区域内，因此颜色充满了整个屏幕。

14.3.3 改正后的代码

为修改该 DLL，必须修改函数 FloodFill()，以确保该点总在构成宽边椭圆的两个椭圆之间。修改上面给出的 DLL 的以下部分，纠正这一问题：

```
__declspec( dllexport ) void WINAPI ThickEllipse(CDC* Pdc,
                                                    int x1,
                                                    int y1,
                                                    int x2,
                                                    int y2,
                                                    int t)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    CBrush newbrush;
    CBrush* oldbrush;
    pDC->Ellipse(x1,y1,x2,y2);
    pdc->Ellipse(x1+t,y1+t,x2-t,y2-t) ;
    Newbrush.CreateSolidBrush( RGB(255,0,0) );
    oldbrush=pDC->SelectObject(&newbrush);
    pDC->FloodFill(x1+(t/2),y1 + ((y2 - y1)/2), RGB(0,0,0));
    pDC->SelectObject(oldbrush);
    newbrush.DeleteObject();
}
```

这一需要调试的问题相对简单，但阐明了调试 DLL 的可能性。

24x7 调试 DLL 期间的问题

调试一个 DLL 时出现的最典型的问题是，在 DLL 中设置的断点不能工作，Microsoft 为此问题提出了几种可能的原因。

首先，当相应的符号信息没有被 Debugger 加载到内存时，可能发现在源文件中不能设置断点。这是因为当相应的符号信息不会被 Debugger 加载到内存时，在源文件中不能设置断点。这一问题可以通过诸如“the breakpoint cannot be set”这样的消息框认出。

当在将要调试的代码前指定的断点已经启动时，Debugger 使用一个断点列表记录如何和在何处设置了断点。然后当调试开始后，Debugger 为代码加载符号信息后执行通过断点列表。它将试图设置断点，但是如果没有为 Debugger 指明任何代码模块，这一努力都将失败。这导致当执行通过断点列表时，没有为 Debugger

处理的符号信息。这一问题的起因包括试图在调用函数 LoadLibrary()之前在 DLL 中设置断点,和在容器启动 ActiveX 服务器之前在该服务器中设置断点。

大多数情况下,可以通过在 Debug/Option 对话框的 Additional DLLs 区域内指定所有另外的 DLL 和 COM 服务器,以提示 Debugger,希望它为另外的 DLL 文件加载符号调试信息避免这一问题。当完成这一任务时,加载到内存的断点将被设置为“虚拟”断点。当代码最终被加载到内存时,它们才变成实际断点。

这一问题的第二个起因是硬盘上有 DLL 的多个副本。当这种情况发生时,尤其是如果在 Windows 目录下,Debugger 就迷惑了。Debugger 将在运行时为指定的 DLL 加载到符号信息(根据 Debug/Option 对话框的 Additional DLLs 区域),而 Windows 可能将 DLL 的一个不同的副本加载到内存。最好的预防措施是在硬盘中只有 DLL 的一个副本,或者确保所有的副本都是同样的。

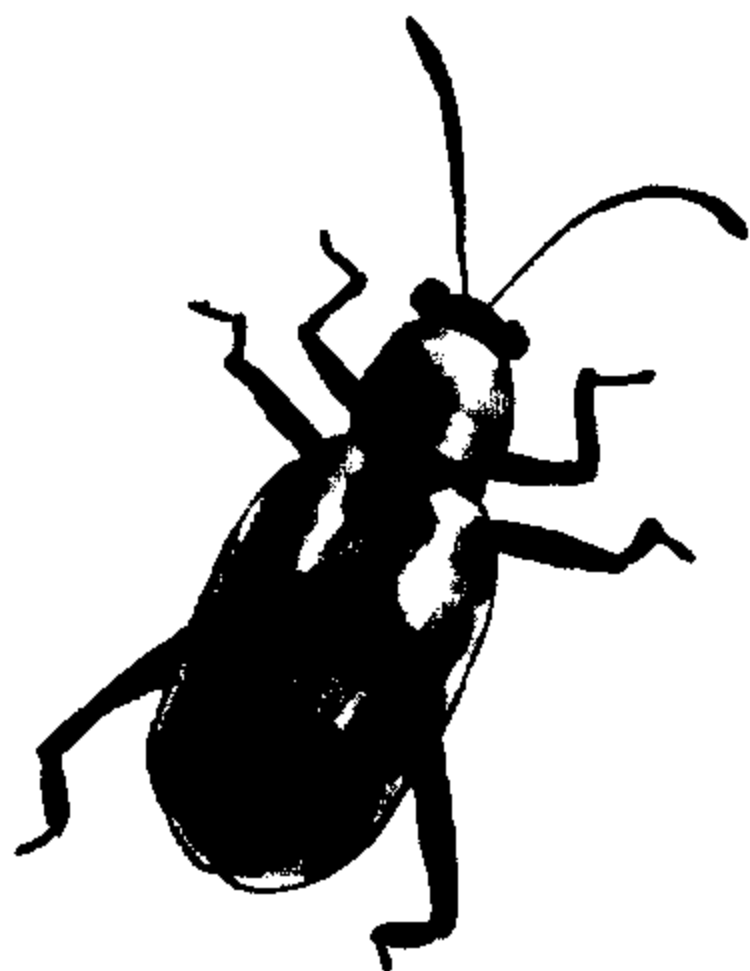
14.4 小结

本章说明了使用主计算机和远程调试计算机时用来调试 DLL 的技术。简单的 DLL(Framer.dll)和主应用程序(DLLDemo.exe)在主计算机上运行,而图形出现在远程调试计算机上。

在两台计算机上调试 DLL 大大增加了先前在第 8 章中介绍的远程调试技术的复杂性。如果进行得慢一些并且慎重一些,确保从头至尾完成所有必需的步骤,那么调试过程应当是很顺利的。 ■

第 15 章

使用 ActiveX 控件工作





所谓 ActiveX 控件, 也被称为 OLE 或定制控件, 是另一个令许多 Visual C++ 程序员感兴趣的话题。ActiveX 控件的普及性和复杂性使 Microsoft 在 Visual C++ 编译器中增加了一个 MFC ActiveX Control Wizard。测试 ActiveX 控件也是一项困难的任务。为帮助测试 ActiveX 控件, Microsoft 增加了有名的 Test Container, 以在构造期间调试控件。

仅仅提出一个带有编码问题的 ActiveX 控件, 然后就说明使用 Debugger 修复其方法, 这对读者显然是不公平的。同时, 我们也认为不应当向熟悉 ActiveX 控件设计的程序员费力地讲解做一个可靠的 ActiveX 控件所必需的所有步骤。

为避免过于简化或过于烦琐, 本章分为两个主要部分。第一部分致力于开发一个简单而实用的 ActiveX 控件。这部分针对有一些 ActiveX 控件经验的程序员。第二部分将说明可以如何调试 ActiveX 控件。这是一个很有趣的部分, 因为读者很可能需要设置两台计算机并远程调试这种类型的工程, 如在其他的 Windows 程序中那样。

15.1 开发一个 ActiveX 控件

Microsoft ActiveX 定制控件是带 OCX 文件扩展名的动态链接库(Dynamic Link Library), 这些为 Windows 98 和 Windows 2000(NT)设计的 32 位控件代替了原来用 Microsoft 的 Visual Basic 开发的 16 位定制控件 VBX。

使用 ActiveX Control Wizard 设计和实现 ActiveX 定制控件相对容易些。为开始创建一个新的工程, 使用 File | New 菜单项打开 New 对话框。为此工程选择 MFC ActiveX Control Wizard 开始设计过程。在设计过程中, Control Wizard 将产生一个缺省的 ActiveX 控件模板, 然后可以根据设计者的指定定制控件。在设计过程中任何时间, 控件都可以在应用程序 Test Container 中测试。

在以下几部分, 将逐步讲解一个名为 Clock 的控件的开发。这一控件与 ActiveX 教学示例中开发的控件类似, 但含有一些定制的特性。

24x7 控件

Windows 将控件分为三类: 标准控件、普通控件和定制控件。标准控件包括单选按钮、按钮、编辑框和列表框。普通控件包括工具栏、工具提示、旋转按钮和滑动杆, 普通控件中的许多已经移植到标准控件工具箱中了。定制控件在某种程度上, 与其他两类控件类似。定制控件的缺点是设计和实现它们需要时间。定制控件的优点是可以制作需要的任何类型的控件, 典型的定制控件包括电子表格、文字处理软件、日历和栅格。

Visual Basic 是第一种将建立定制控件过程变得容易的语言, 这些定制控件很快就被称为 VBX 控件, 因为其文件扩展名是.vbx。定制控件实际上是小的动态链接库(DLL)。

开始时, Visual C++ 程序员不得不采用或创建 Visual Basic 定制控件再结合到自己的应用程序中。以后

产生了 Visual C++ 编译器，定制控件可以在 C++ 环境下开发。

对于 ActiveX 控件开发人员来说最好的消息是 Visual C++ 编译器提供了一个 ControlWizard 以帮助产生定制控件并结合到应用程序中。Visual C++ 编译器还提供了一个专用应用程序 Test Container，以在创建定制控件期间进行测试。

使用 ControlWizard 的好处是，ActiveX 控件的模板将是“24x7”的。换句话说，最初的代码已经被彻底地测试并调试过。

设计提示

ActiveX 控件毕竟仅是控件。例如，它们可以在对话框中使用，然后放到标准控件和普通控件中，然而设计和实现定制控件要困难得多。作为控件开发人员，必须生成、编写和编译绘制定制控件的代码，然后实现控件的所有方法等等，这一定制控件代码将变成一个动态链接库(DLL)。ActiveX 控件使用文件扩展名 OCX，而不是前面一章讨论的动态库的文件扩展名 DLL。然后使用该定制控件的应用程序必须与控件的方法、数据等相互作用，这些代码也必须有程序员编写。ActiveX 控件必须完全可重入。由于这些控件是真正独立的 DLL，所以它们不与应用程序链接。这样 ActiveX 控件要求控件的每一次使用都有一个单独的数据实例，消息是在 ActiveX 控件和应用程序之间唯一允许的通信方式。

使用 MFC ActiveX ControlWizard 时，这些困难的任务中的大部分是自动完成的。

15.1.1 使用 ControlWizard

使用 File | New 菜单选项，打开 New Project 对话框，开始设计过程。选择 MFC ActiveX ControlWizard 项，然后命名工程为 Clock，如图 15-1 所示。

在 ControlWizard 的设计过程的第一步，为此工程选择图 15-2 中所示各项。

图 15-3 显示了在定制控件设计过程中可以设置或修改的其他工程选项。

然后为此控件使用缺省选项，只是有一个例外，即选择 Available in “Insert Object” dialog 复选框。这将允许此控件被诸如 Word、Access 和 Excel 这样的 OLE 应用程序注册和使用。单击 Finish 按钮，完成设计过程。

图 15-4 显示了在最终的工程中包含的组件的列表。

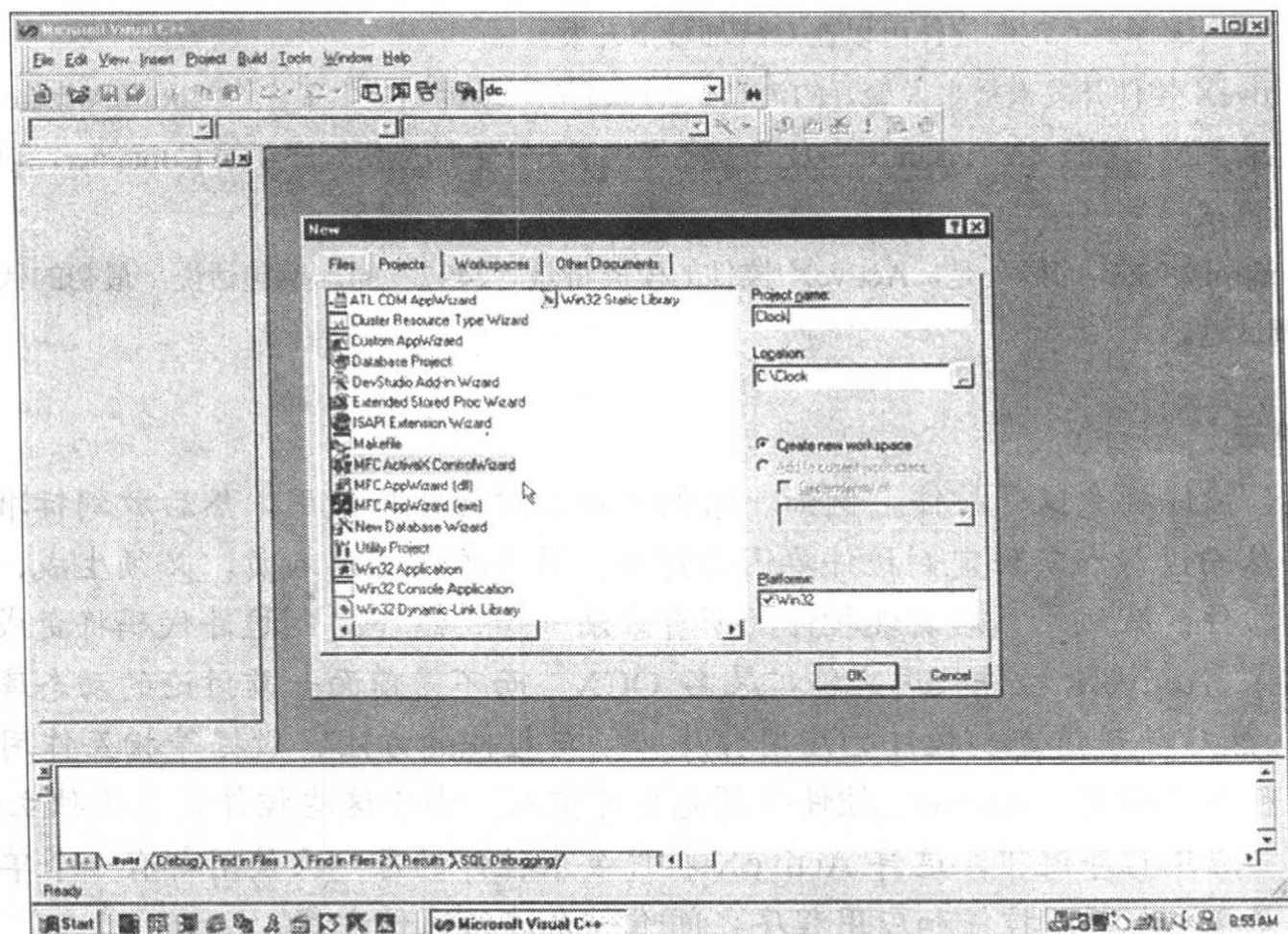


图 15-1 选择 MFC ActiveX Control Wizard 开始控件设计过程

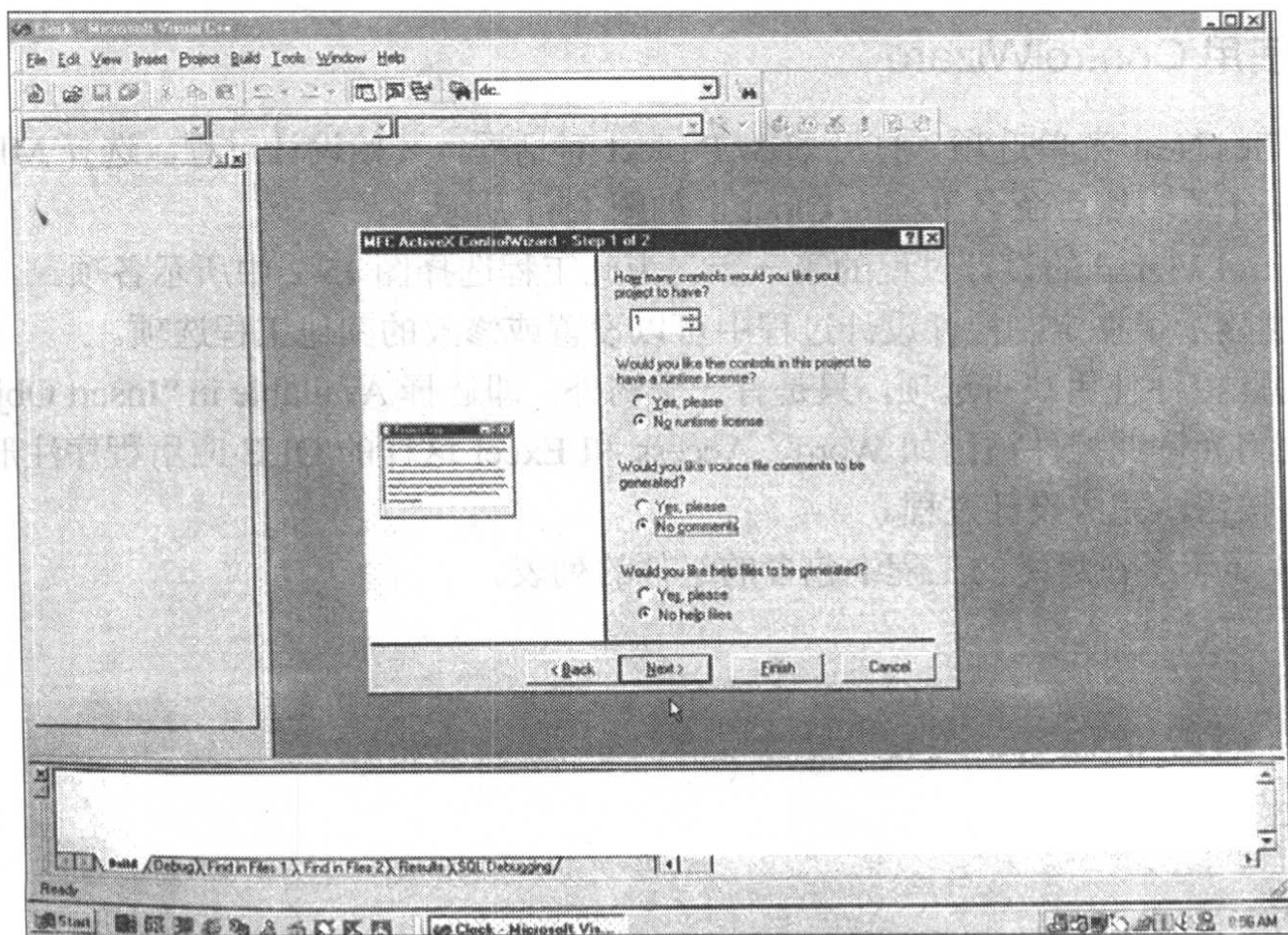


图 15-2 建立一个 ActiveX 控件的第一步

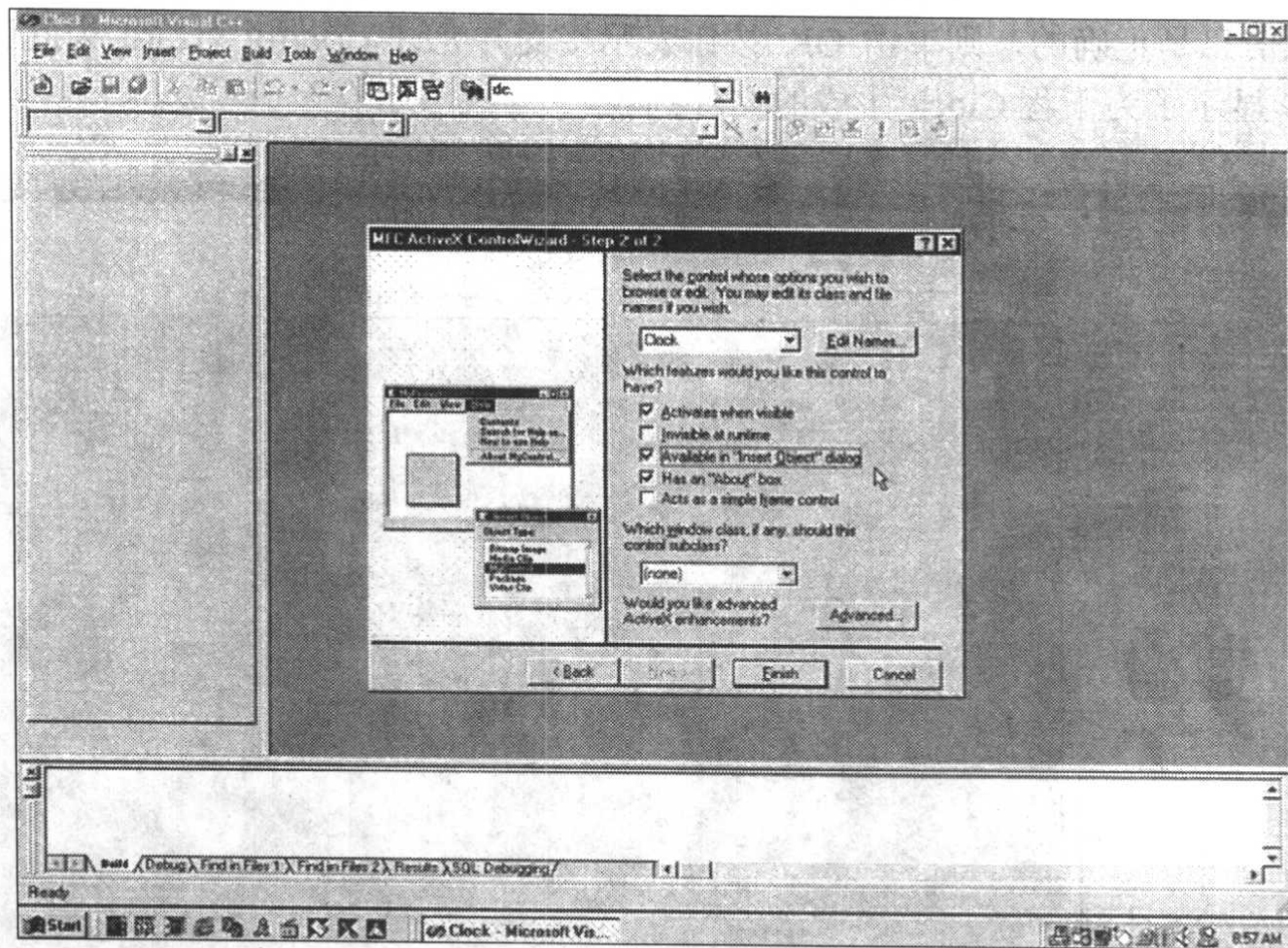


图 15-3 控件建立过程的第二步

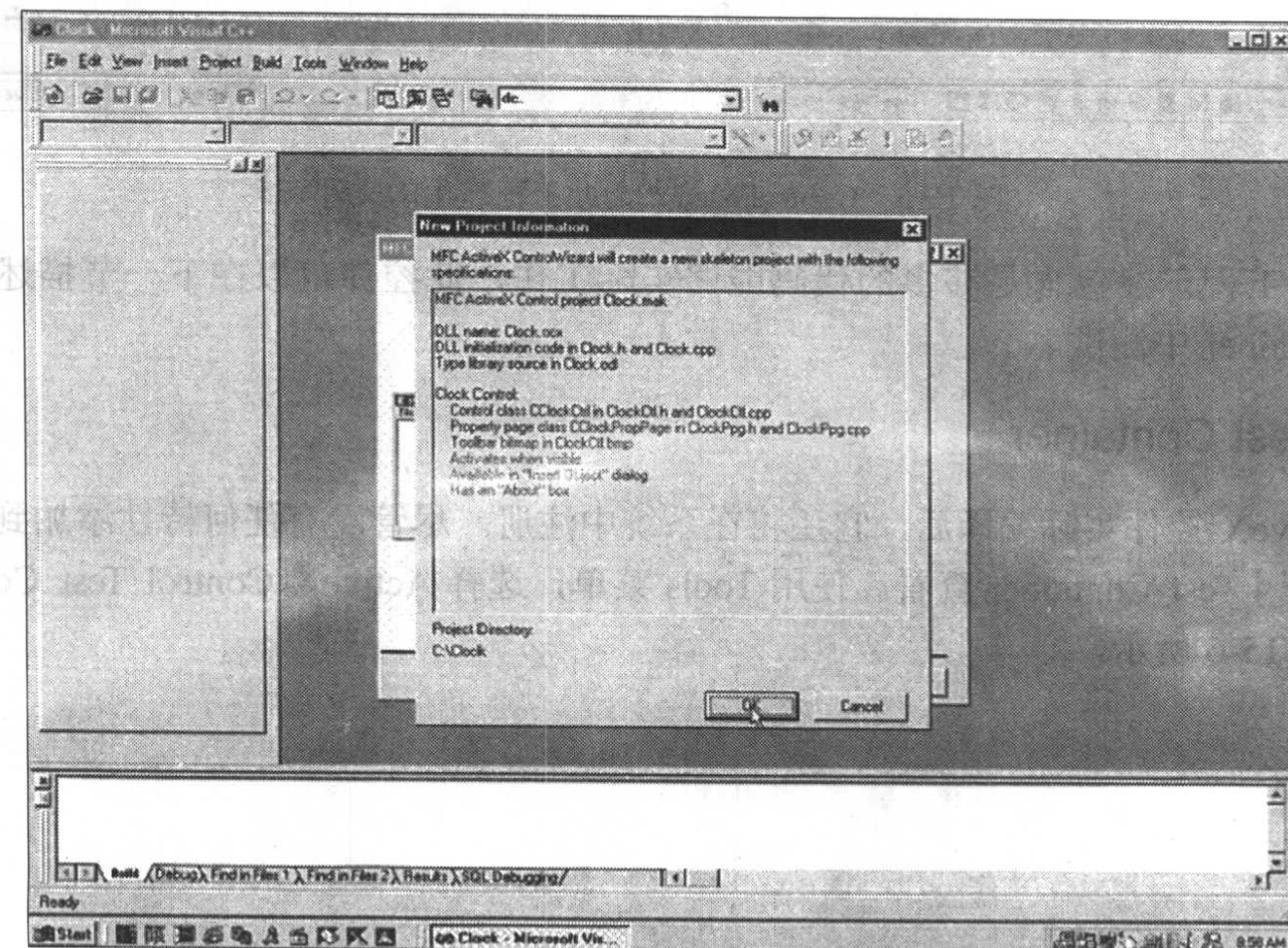


图 15-4 控件的工程信息



如果此信息是正确的,则单击 OK 按钮表示接受并允许 ControlWizard 为此工程生成文件。图 15-5 显示了为工程 Clock 生成的文件列表。

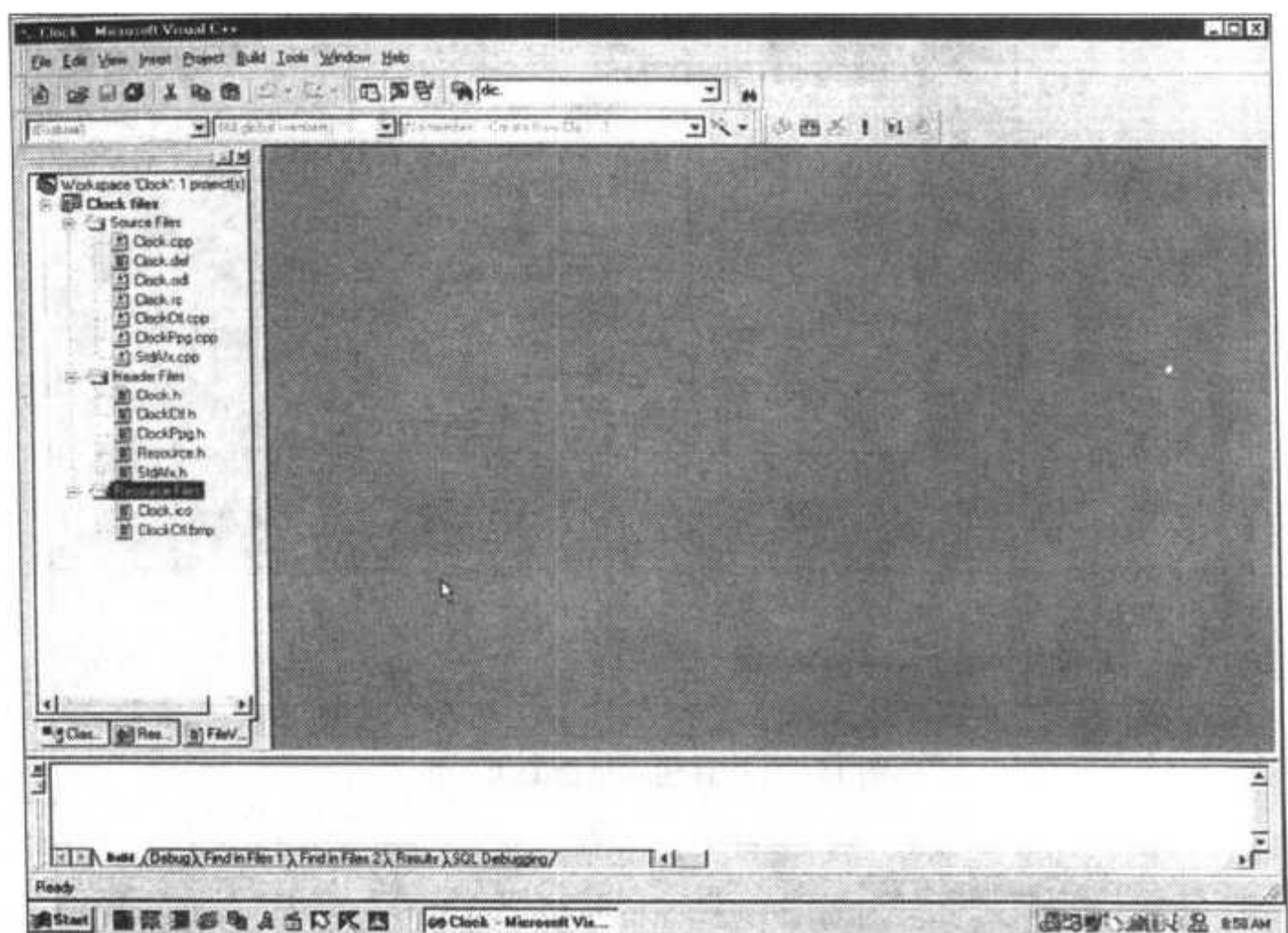


图 15-5 ControlWizard 为此工程产生的文件

此时,工程已经包含足够多的代码能够实际工作,此控件可以在下一节描述的应用程序 Test Container 中测试。

15.1.2 Test Container

当 ActiveX 控件实际编译后,它还将在系统中注册。尽管没有任何特性添加到控件中,它也可以使用 Test Container 查看。使用 Tools 菜单,选择 ActiveX Control Test Container 菜单项,如图 15-6 所示。

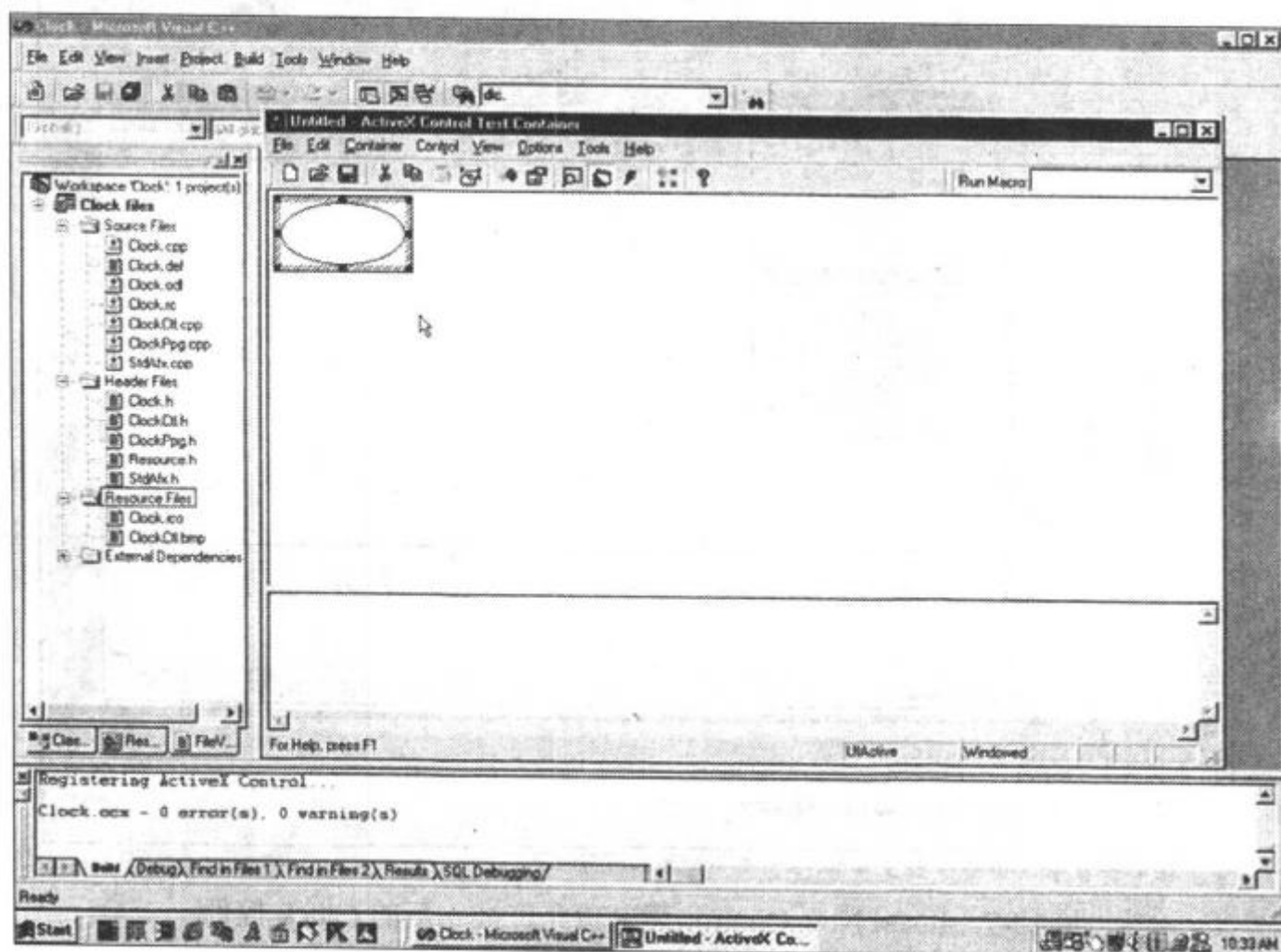


图 15-6 使用 Control Test Container 检查缺省控件

设计提示

在一个新的 ActiveX 控件的开发和测试过程中经常使用 Control Test Container。除了使用容易之外，Test Container 还与 Debugger 全面结合。使用 Debugger 时，Test Container 可以自动启动并且甚至在远程调试状态下使用。

Test Container 启动后，客户区域是空的。打开 Edit 菜单后选择 Insert New Control 菜单项，如图 15-7 所示。

选择 Insert New Control 菜单项后，将显示一个可能的控件的列表，如图 15-8 所示。

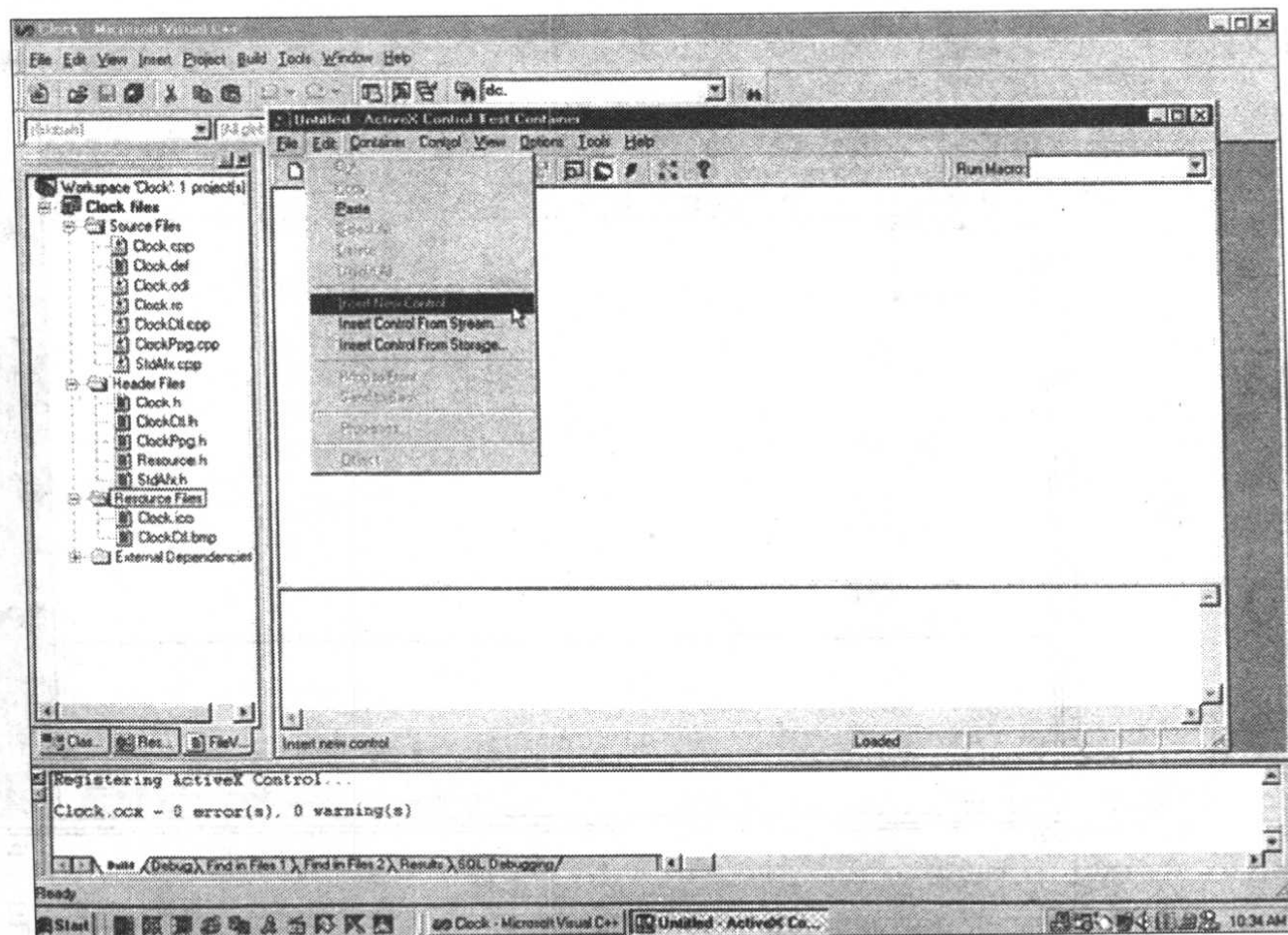


图 15-7 Insert New Control 菜单项允许插入新的 Clock 控件

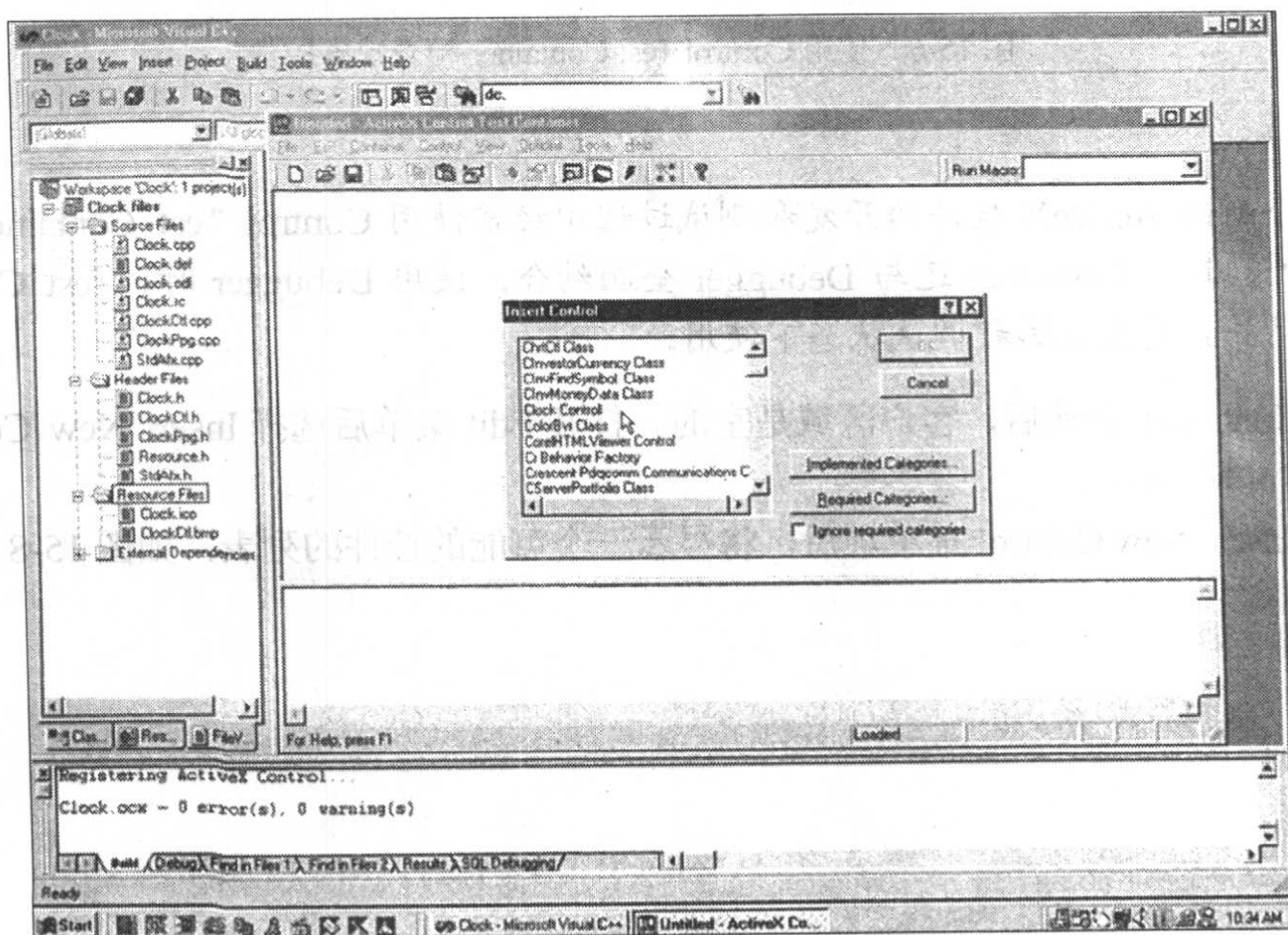


图 15-8 注册控件列表中包含新的 Clock 控件

图 15-9 显示了从列表中选择 Clock 控件后的 Test Container。

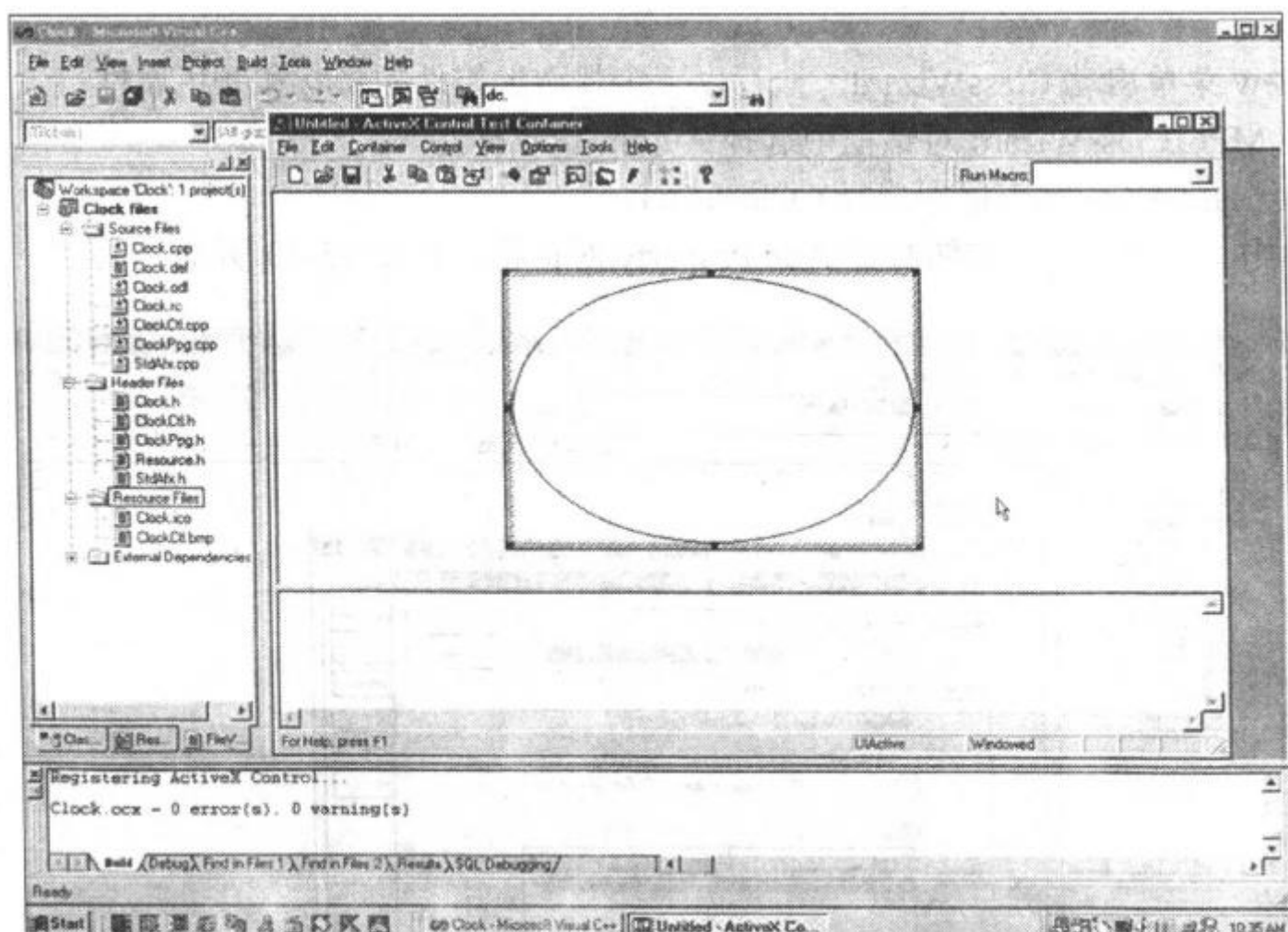


图 15-9 Clock 控件被插入到 Test Container 中

所有用 ControlWizard 开发的控件初始时都是这个样子。按照缺省设定，这一控件是一个被边界矩形包围的椭圆。

15.1.3 产生一个真实的 Clock 控件

ClassWizard 用来修改 ControlWizard 产生的缺省定制控件。

为建立一个真实的自定义 Clock 控件，从前面一节描述的缺省定制控件着手。此处是我们实际 Clock 控件的特性：

- 钟面总是圆形的，而不能是椭圆形的。
- 钟面和背景是同一颜色的。
- 响应发生在控件内的鼠标事件，并报告当前系统日期和时间。

可以通过仅处理工程的两个文件 ClockCtl.cpp 和 ClockCtl.h，添加所有这些新的特性到控件中。在下一部分，我们将添加一些新的特性。



15.1.3.1 改变钟的形状、大小和颜色

从 View 菜单启动 ClassWizard，执行以下步骤添加控件的形状和颜色属性：

1. 在 MFC ClassWizard 对话框中选择 Automation 标签。
2. 从 Class name 列表框中选择 CClockCtrl。
3. 单击 Add Property 按钮打开 Add Property 对话框，如图 15-10 所示。

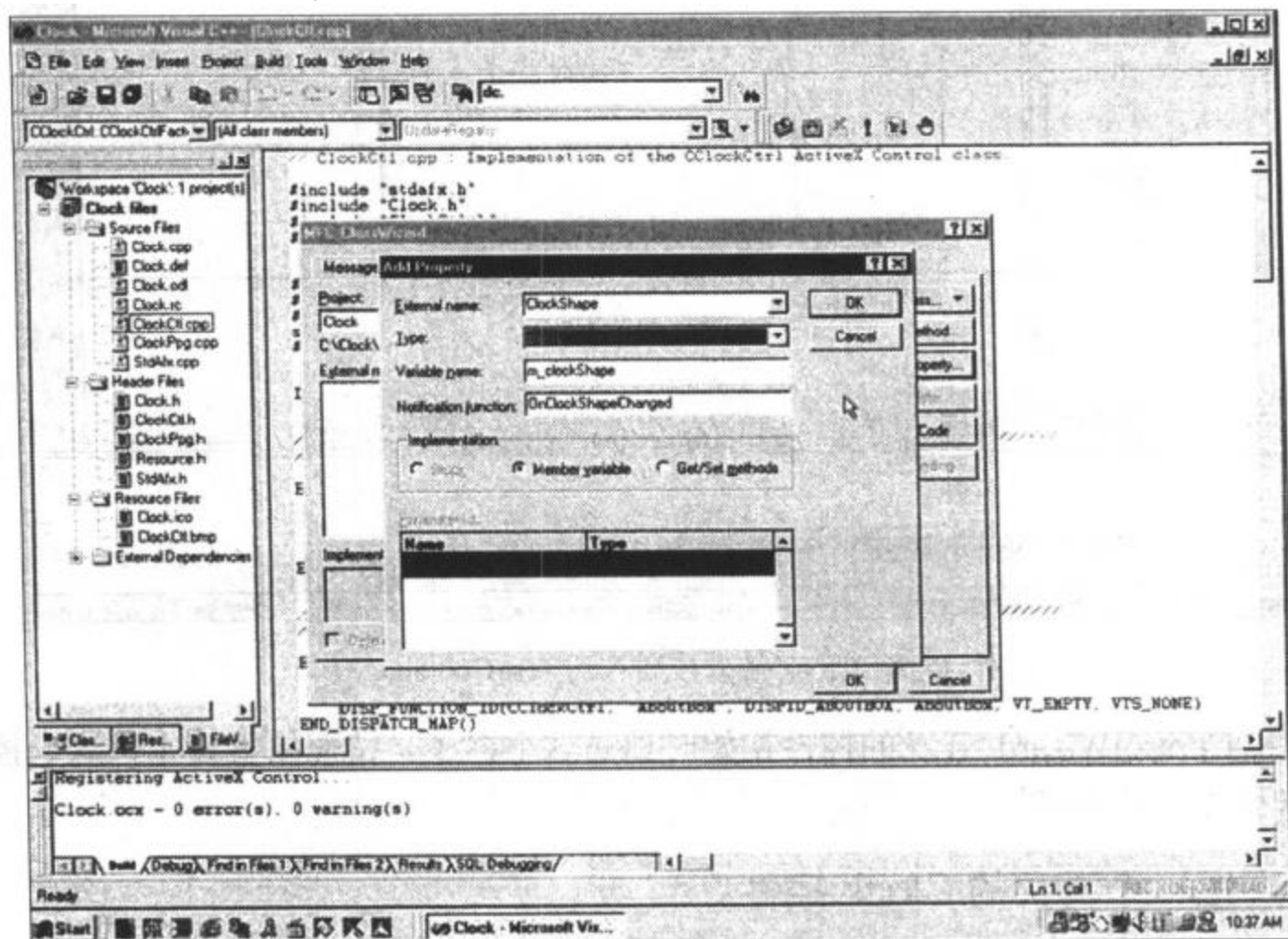


图 15-10 使用 Add Property 对话框添加 ClockShape 属性

4. 输入名字 ClockShape 作为 External name。
5. 为 Implementation 选择 Member variable。
6. 从 Type 下拉列表框中选择 BOOL。注意 Notification function 编辑控件中含有 OnClockShapeChanged，成员变量是 m_clockShape。
7. 单击 OK 按钮接受这些值，回到 Automation 标签。
8. 再次单击 Add Property 按钮打开 Add Property 对话框。
9. 在 External name 组合框的编辑控件中，从下拉列表中选择 BackColor。
10. 为 Implementation 选择 Stock。

11. 单击 OK 按钮接受这些值，回到 Automation 标签，现在的 MFC ClassWizard 对话框应当与图 15-11 相似。

12. 单击 OK 按钮接受这些选择，关闭 ClassWizard。

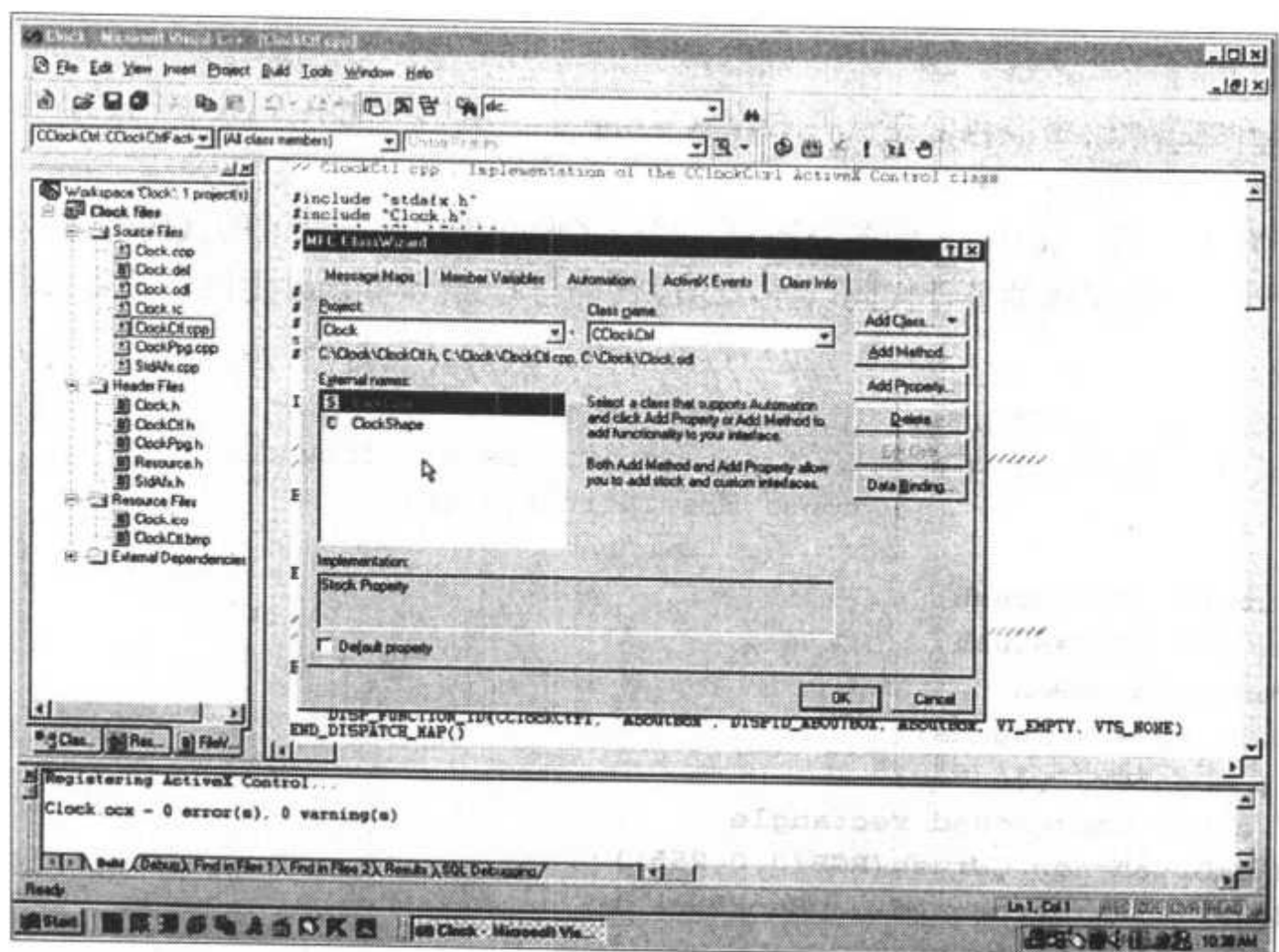


图 15-11 ClassWizard 添加属性 BackColor 和 ClockShape 到工程中

ClassWizard 将在 CClockCtrl 类中添加属性 ClockShape 和 BackColor，CClockCtrl 类的派遣映射将被改变以适应 ClockShape 属性，而 BackColor 属性将被添加到库文件 clock.odl 中，函数 OnClockShapeChange() 的一个声明将被添加到头文件 ClockCtl.h 中。

ClassWizard 自动添加上面讨论的改变。现在我们的工作编写相应的代码。

头文件 ClockCtl.h

编写一个函数确定钟面的正确尺寸，这一函数将被称为 GetDrawRect()。在头文件 ClockCtl.h 中将此函数的声明放在析构函数声明的后边，见下面的部分清单：

```
// Implementation
protected:
    ~CClockCtrl();
    .
    .
    .
```



```
void GetDrawRect(CRect* rc); // Clock Size
```

```
DECLARE_OLECREATE_EX(CClockCtrl) //Class factory and guid
```

这一函数的代码将被包含在文件 ClockCtl.cpp 中，见下一部分。

文件 ClockCtl.cpp

以下为文件 ClockCtl.cpp 的部分清单，显示了此文件中必须修改的代码部分。除了添加或修改控件属性的黑体的各行代码以外，这段代码与 ControlWizard 返回的缺省文件相同。

```
////////////////////////////////////  
// CClockCtrl::OnDraw - Drawing function  
void CClockCtrl::OnDraw(CDC* pdc, const CRect& rcBounds,  
                        const CRect& rcInvalid)  
{  
    CBrush* pBackBrush;  
    CBrush* pFaceBrush;  
    CPen* pHandsPen;  
    CRect rc = rcBounds;  
    int xcenter, ycenter;  
    // clock background rectangle  
    pBackBrush=new CBrush(RGB(0,0,255));  
    pdc->FillRect(rcBounds, pBackBrush);  
    GetDrawRect(&rc);  
    // clock face  
    pFaceBrush=new CBrush(RGB(255,255,0));  
    pdc->SelectObject(pFaceBrush);  
    pdc->Ellipse(rc);  
    // clock hands  
    pHandsPen=new CPen(PS_SOLID,3,RGB(0,0,0));  
    pdc->SelectObject(pHandsPen);  
    xcenter=(rc.right-rc.left)/2 + rc.left;  
    ycenter=(rc.bottom-rc.top)/2 + rc.top;  
    pdc->MoveTo(xcenter,ycenter);  
    pdc->LineTo(rc.right*75/100,rc.bottom*75/100);  
    pdc->MoveTo(xcenter,ycenter);  
    pdc->LineTo(xcenter,rc.bottom*90/100);  
    delete pBackBrush;  
    delete pFaceBrush;  
    delete pHandsPen;  
}
```

```

.
.
.
void CclockCtrl::GetDrawRect(Crect* rc)
{
    // Round clock face
    int dx = rc->right - rc->left;
    int dy = rc->bottom - rc->top;
    if(dx>dy){
        rc->left += (dx - dy) / 2;
        rc->right = rc->left + dy;
    }
    else {
        rc->top += (dy - dx) / 2;
        rc->bottom = rc->top + dx;
    }
}
}

```

钟面是用函数 `ellipse()`画出的，然后用黄色笔刷填充：

```

// clock face
pFaceBrush=new CBrush( RGB(255,255,0) );
pdc->SelectObject(pFaceBrush);
pdc->Ellipse(rc);

```

为了避免由于拉伸控件而得到一个椭圆形的钟，应测试控件。使用函数 `GetDrawRect()` 确定的矩形的 x 方向和 y 方向边长的较小者，作为钟面的直径。这一信息是由参数 `rc` 传递给函数 `Ellipse()`的，此参数保存了边界矩形的坐标。

表针是结合函数 `MoveTo()`和 `LineTo()`画出的：

```

// clock hands
pHandsPen=new CPen(PS_SOLID,3,RGB(0,0,0));
pdc->SelectObject(pHandsPen);
xcenter=(rc.right-rc.left)/2 + rc.left;
ycenter=(rc.bottom-rc.top)/2 + rc.top;
pdc->MoveTo(xcenter,ycenter);
pdc->LineTo(rc.right*75/100,rc.bottom*75/100);
pdc->MoveTo(xcenter,ycenter);
pdc->LineTo(xcenter,rc.bottom*90/100);

```

在画表针时距钟面的中心有一个成比例的距离。



15.1.3.2 响应鼠标事件

控件 Clock 必须响应鼠标事件以显示当前日期和时间。当单击钟面时，钟 Clock 将改变颜色，并为控件报告系统时间。颜色变化和时间信息表明发生了一个控件“命中”(hit)。

控件命中可以通过以下步骤添加到工程中：

1. 在 MFC ClassWizard 对话框中选择 Automation 标签。
2. 在 Class name 列表框中选择 CClockCtrl。
3. 单击 Add Property 按钮打开 Add Property 对话框。
4. 在 External name 组合框的编辑框中，输入 HitClock。
5. 确保为 Implementation 选中了 Member Variable。
6. 在 Type 列表框中选择 OLE_COLOR 后清空 Notification function 编辑框。
7. 单击 OK 按钮关闭 Add Property 对话框并回到 Automation 标签。屏幕应当与图 15-12 所示相似。

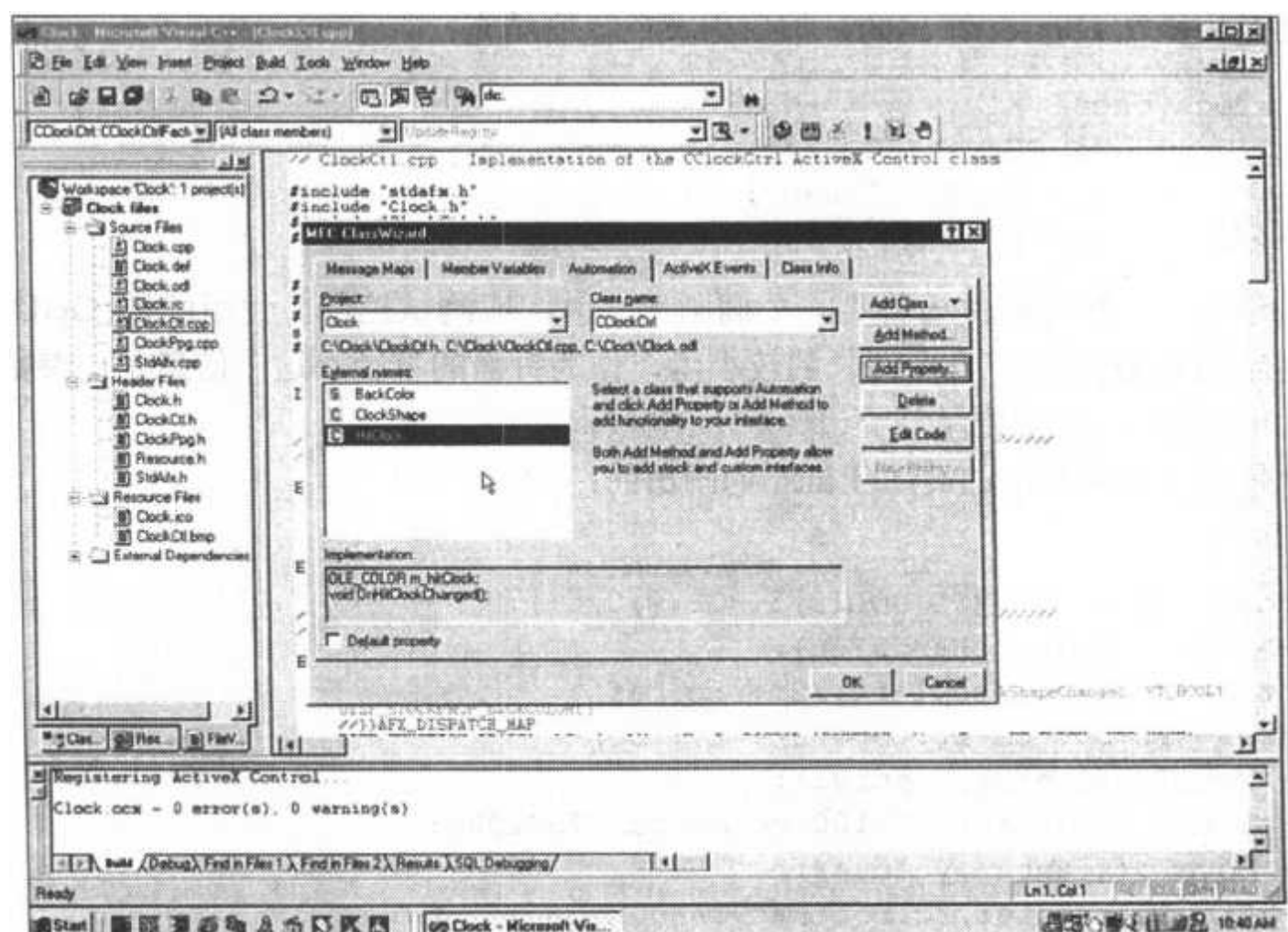


图 15-12 属性 HitClock 允许控件响应鼠标事件

8. 选择 Message Maps 标签。

9. 在 Class name 列表框中选择 CClockCtrl。
10. 在 Object IDs 列表框中选择 CClockCtrl，然后查看 Message 列表框中的消息列表。
11. 在 Message 列表框中选择 WM_LBUTTONDOWN。
12. 单击 Add Function 按钮。
13. 选择 WM_LBUTTONUP，重复以上的过程。最终的屏幕看起来应当如图 15-13 所示。
14. 单击 OK 按钮接受这些选择，关闭 ClassWizard。

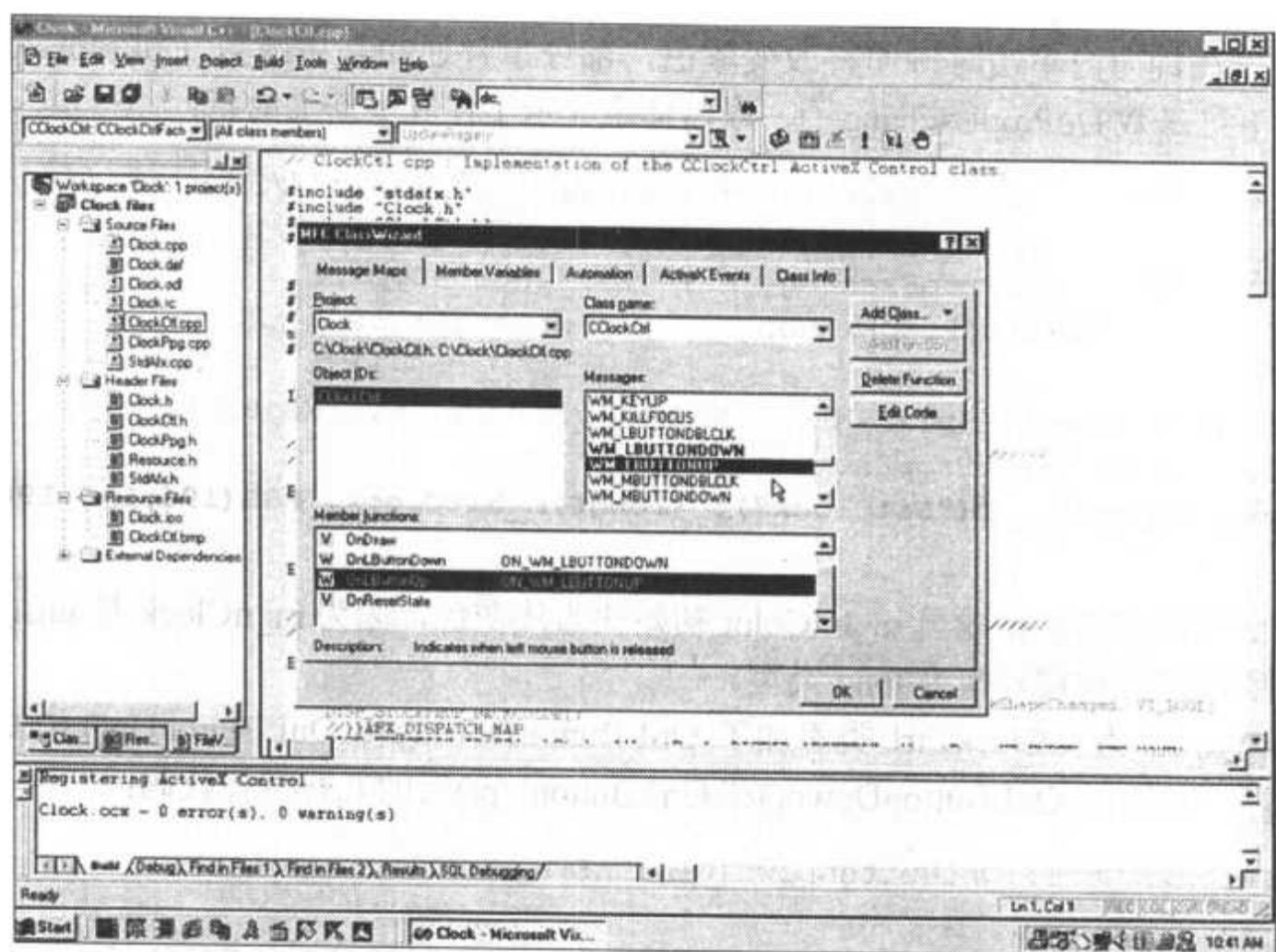


图 15-13 两个成员函数将响应鼠标按钮事件

ClassWizard 将自动为类 CClockCtrl 添加 HitClock 属性和先前成员函数实现的框架。再次，这些改变是由 ClassWizard 自动添加的，下面编写与这些事件相对应的代码。

头文件 ClockCtl.h

为适应两种新的方法，必须在头文件 ClockCtl.h 中插入另外的两个方法 Inface()和 HitClock()。第一种方法用来确定在 Clock 控件内是否有鼠标事件发生，第二种方法用来在命中发生时改变钟面的颜色。在下面的头文件 ClockCtl.h 部分清单中列出的析构函数的后边插入了 InFace()和 HitClock():



```
// Implementation
protected:
~CClockCtrl();
void GetDrawRect(CRect* rc);    //Clock Sizi
BOOL InFace(CPoint& point);    //Hit the Clock?
void HitClock(CDC* pdc);      //Blink the clock color
DECLARE_OLECREATE_EX(CClockCtrl) //Class factory and guid
```

下面在源文件中添加代码以检测控件内的单击。

文件 ClockCtrl.cpp

当用户在钟面内单击时钟面将改变颜色，部分事件通知是由函数 DoPropExchange()处理的。以下是函数 DoPropExchange()，以黑体字显示了在其中新加的行：

```
////////////////////////////////////
////////////////////////////////////
// CClockCtrl::DoPropExchange - Persistence support
void CClockCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);
    px_Long(pPX, _T("HitClock"), (long&)m_hitClock, RGB(196,196,196));
}

```

这一函数负责将数据成员 m_hitColor 初始化为浅灰色，因为 m_nClock 是 unsigned long 类型的，所以它必须被转换为 long 类型。

ClassWizard 为 CClockCtrl 类添加了 OnLButtonDown()和 OnLButtonUp()方法。下面的清单显示了添加方法 OnLButtonDown()和 OnLButtonUp()后的几行黑体代码：

```
void CClockCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add message handler code here and/or call default
    CDC* pdc;
    // Flash a color change for clock face
    if (InFace(point)) {
        pdc = GetDC();
        HitClock(pdc);
        ReleaseDC(pdc);
    }
    COleControl::OnLButtonDown(nFlags, point);
}
void CClockCtrl::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add message handler code here and/or call default

```

```

    if (InFace(point)) {
        InvalidateControl();
        COleControl::OnLButtonUp(nFlags, point);
    }

```

这段代码检查在钟面内是否单击。如果单击，函数 HitClock() 将被调用改变钟面的黄色变浅灰色。

单击后，函数 OnLButtonUp() 使控件无效，将钟面刷新为黄色。

函数 InFace() 用来确定单击是否是在钟面内完成的，必须在 ClockCtl.cpp 清单的结尾添加以下所有代码：

```

BOOL CClockCtrl::InFace(CPoint& point)
{
    CRect rc;
    GetClientRect(rc);
    GetDrawRect(&rc);
    // Find center point
    double h = (rc.right - rc.left) / 2;
    double k = (rc.bottom - rc.top) / 2;
    // Find x and y values
    double x = point.x - (rc.right + rc.left) / 2;
    double y = point.y - (rc.bottom + rc.top) / 2;
    // Ellipse equation determines location of point
    return ((x * x) / (h * h) + (y * y) / (k * k) <= 1);
}

```

InFace() 函数定位 Clock 控件的中心并确定在钟面内是否发生命中。

如果该点落在钟面内，函数 HitClock() 将被调用。下面函数中的所有代码必须被添加到 ClockCtl.cpp 清单的结尾处：

```

void CClockCtrl::HitClock(CDC* pdc)
{
    CBrush* pOldBrush;
    CBrush hitBrush(TranslateColor(m_hitClock));
    CRect rc;
    TEXTMETRIC tm;
    Struct tm *date_time;
    time_t timer;
    // Fill between text
    / Background mode to transparent
    pdc->SetBkMode(TRANSPARENT);
    GetClientRect(rc);
    pOldBrush=pdc->SelectObject(&hitBrush);
    pdc->Ellipse(rc);
    // Get time and date

```



```
time(&timer);  
date_time=localtime(&timer);  
const CString& strtime = asctime(date_time);  
// Get Font information then print  
pdc->GetTextMetrics(&tm);  
pdc->SetTextAlign(TA_CENTER | TA_TOP);  
pdc->ExtTextOut((rc.left + rc.right)/2,  
               (rc.top + rc.bottom - tm.tmHeight)/2,  
               ETO_CLIPPED, rc, strtime,  
               strtime.GetLength()-1, NULL);  
pdc->SelectObject(pOldBrush);  
}
```

HitClock()函数中的代码选用前面定义的灰色笔刷刷新整个 Clock 区域。

现在完成了代码，准备在 Control Test Container 中测试。按照在本章前面讨论的步骤，将 Clock 控件加载到 Test Container。在钟面内单击，看到的屏幕类似于图 15-14 所示。

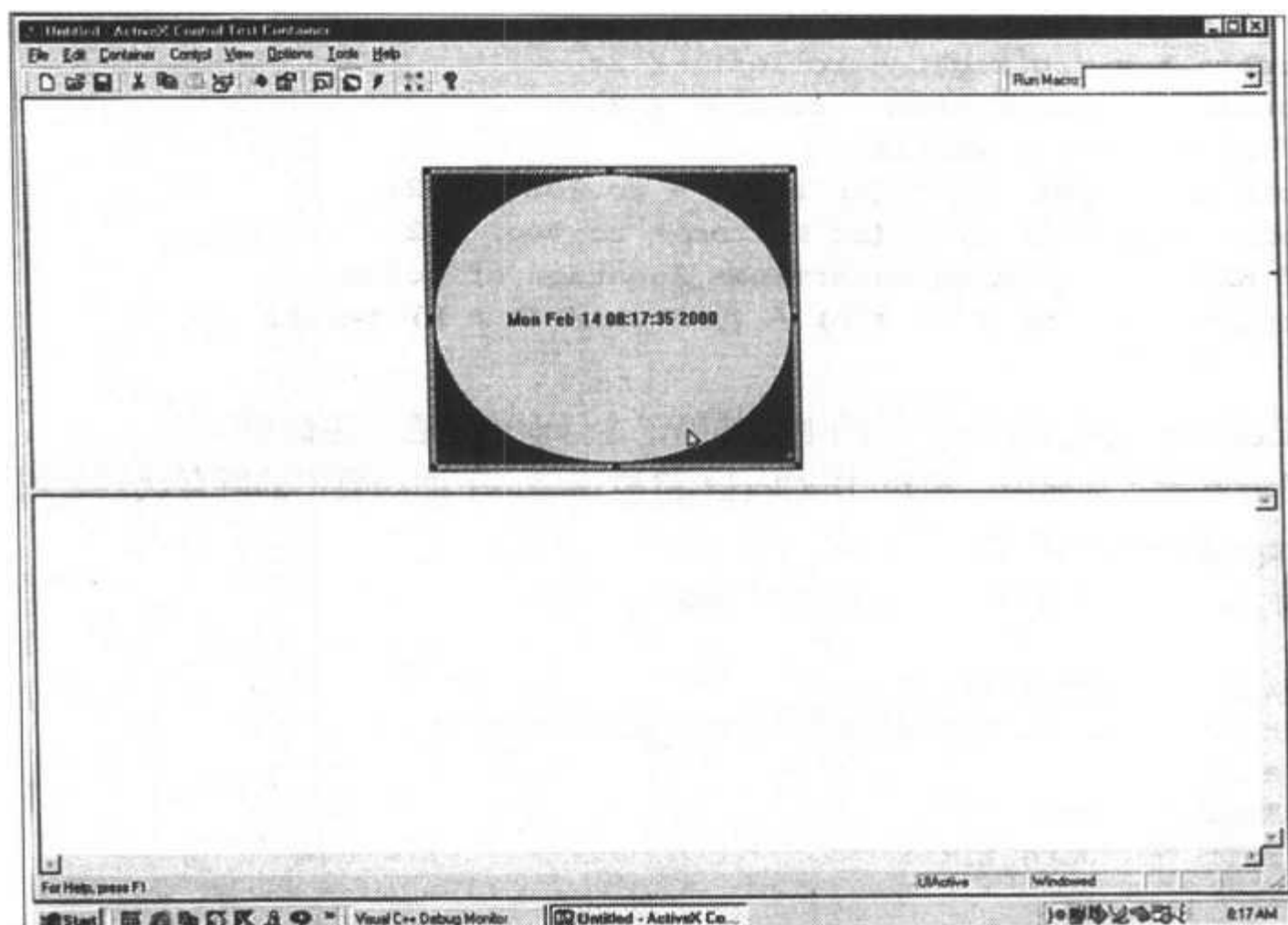


图 15-14 在钟面上单击时产生的结果

控件似乎有一个问题。我们将在 Debugger 和 Control Test Container 的帮助下利用远程计算机调试功能调查这一问题。

15.2 调试 Clock 控件

在上一节，我们发现控件没有产生希望的结果。钟面开始时看来还是好的，而当单击时它就由圆形变成了一个椭圆。然而，一旦结束单击，钟面又变回为圆形。为了查清这一问题，我们将使用远程调试以在单步调试控件的代码时始终看到控件。在下一部分，将介绍如何设置计算机，以在主计算机上操作 Debugger 的同时在远程计算机上使用 Control Test Container。

15.2.1 准备远程目标计算机

回想第 8 章，远程目标计算机上必须运行一个名为 Remote Debug Monitor 的小程序。我们曾为此目的使用了一台 Sony 便携式计算机。这一应用程序和这台计算机负责与主计算机上运行的 Debugger 通信。在我们的例子中，主计算机是 hp 的。软件 Remote Target Computer 控制 Debugger 中应用程序的执行。

为安装 Remote Debug Monitor，远程目标计算机(Sony)必须包含一些其他文件。对于 Windows 98 和 Windows NT(2000)，这些文件包括：

```
MCVCMON.EXE
MSVCRT.DLL
TLNOT.DLL
DM.DLL
MSVCP60.DLL
MSDIS110.DLL
PSAPI.DLL (仅对 NT)
```

使用计算机 Start 窗口的 Find 选项在主计算机(hp)的正确子目录中查找这些文件。将这些文件从主计算机(hp)复制到远程目标计算机(Sony)上，并且保存在远程目标计算机的 Windows 子目录中。唯一的例外是文件 MSVCRT.DLL 应当复制到子目录 Windows\System32 中。一旦完成这些，重新启动远程目标计算机。

为在远程目标计算机(Sony)上运行软件 Remote Target Computer，执行下列步骤：

1. 在远程目标计算机(Sony)上运行应用程序 MCVCMON.EXE。
2. Visual C++ Debug Monitor 对话框出现。
3. 选择 Settings 选项。
4. Win32 Network Settings 对话框出现。
5. 在这一对话框中输入主计算机的名字(在本例中为 hp，也可以使用 IP 地址。)
6. 如果口令文本框激活，输入一个口令。这一口令在两台计算机中必须匹配。否则，



也可以为空。

7. 单击 OK 按钮。
8. 单击 Connect 按钮。

这 8 步一旦完成，屏幕上将出现 Connecting 对话框。这时不执行任何操作。一旦实际调试开始，这一对话框将自动消失。完成调试过程以后，单击 Disconnect 按钮终止远程连接。

15.2.2 准备主计算机

主计算机(hp)必须准备与远程目标计算机(Sony)通信。

在主计算机(hp)上执行以下步骤：

1. 在 Visual C++ 中，选择 Build | Debugger Remote Connection 菜单项。
 2. 出现 Remote Connection 对话框。
 3. 如果 Platform 下拉列表允许选择，则选择正确的平台。如果没有提供任何选项，那么缺省项被自动选择。
 4. 使用 Connection 下拉列表选择 Network(TCP/IP)连接选项。
 5. 选择 Settings 选项。
 6. 出现 Win32 Network (TCP/IP) Settings 对话框。
 7. 在此对话框中输入远程目标计算机(Sony)的名字。也可以使用 IP 地址。如果有口令选项，则输入一个与远程目标计算机相同的口令。在远程计算机和主机中，这一框也可以为空。
 8. 单击 OK 按钮关闭 Win32 Network Settings 对话框。
 9. 单击 OK 按钮关闭 Remote Connection 对话框。
- 这 9 步一旦完成，则设置完成主计算机就可以与远程目标计算机通信。

15.2.3 开始调试过程

当网络中的两台计算机都为通信做好准备之后，调试过程就可以开始了。记住，运行 Visual C++ 的主计算机是 hp 计算机，而远程目标计算机是 Sony 计算机。完成以下建议的步骤：

1. 全部复制要调试的工程的子目录到两台计算机中，使用同样的目录名称。
2. 在网络中，这两个目录都应当设置为共享目录。
3. 启动 Visual C++ 编译器，将工程加载到主计算机。
4. 选择 Project | Settings 菜单选项。出现 Project Settings 对话框。
5. 选择 Project Settings 对话框中的 Debug 标签。
6. 从 Category 列表中选择 General，然后设置以下各项：
 - Category 列表框：输入工程所需要的任何其他 DLL(即 Additional DLLs)。

- Executable for Debug Session 编辑框：输入 Control Test Container 的名称和路径。这也可以从控件中选择，如图 15-15 所示。
 - Working Directory 文本框：空。
 - Program Arguments 文本框：空，除非程序接受初始参数。
 - Remote Executable Path 文本框：在远程目标计算机(Sony)上输入 Control Test Container 的名称和路径。
 - 选择 Build | Start Debug 菜单项，然后使用 Go(F5)选项到达一个预设的断点。
7. 给系统至少一分钟接通网络，然后以正常方式开始调试过程。Test Container 将自动启动。
8. 记住，每在工程中做一次改动，两台机器上的文件都必须更新。
9. 使用 Test Container 的 Edit 菜单从提供的注册控件列表中选择 Clock 控件，将其添加到远程计算机上的 Test Container 中。

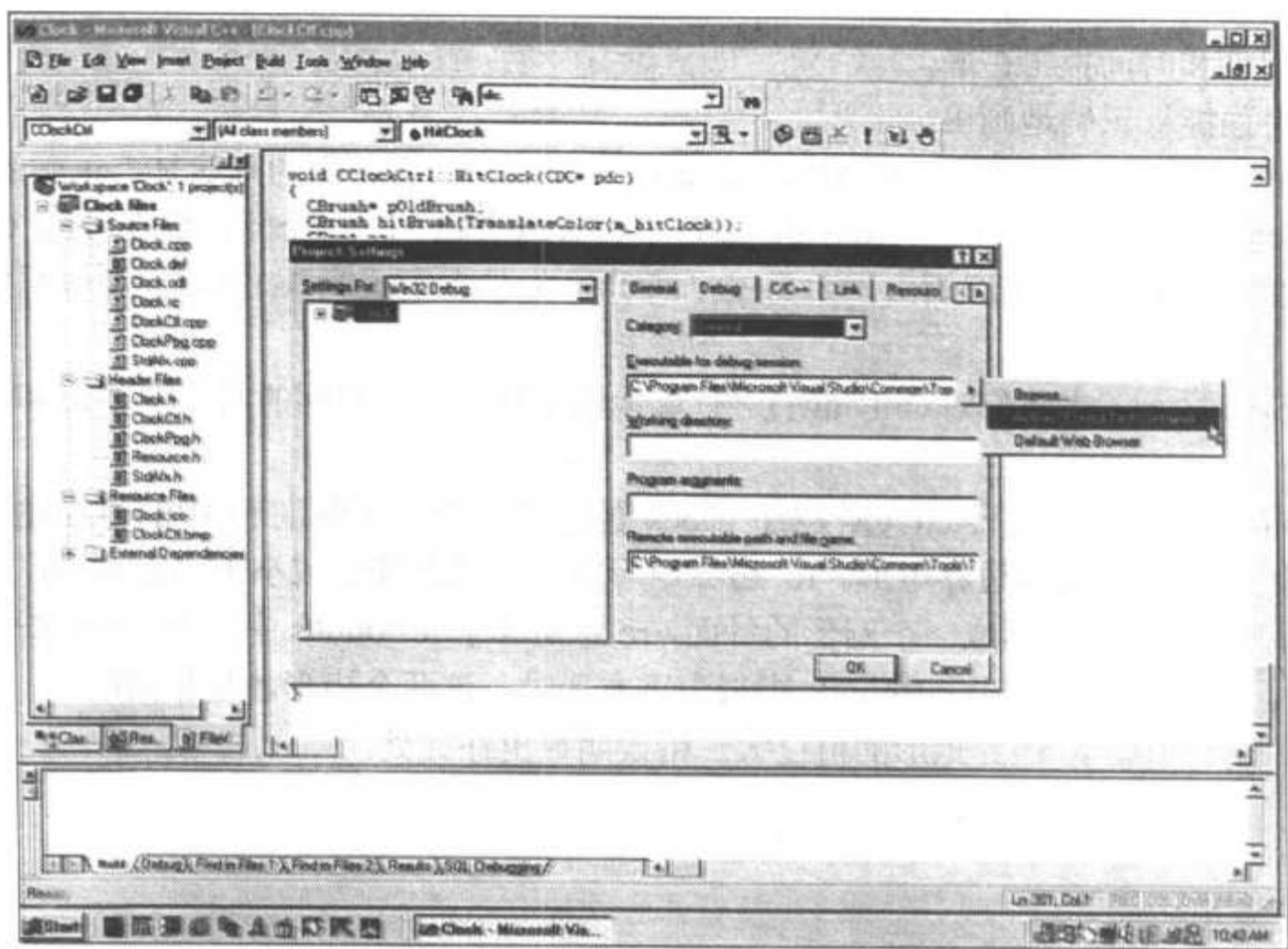


图 15-15 主计算机(1p)上显示的 Project Settings 对话框

设计提示

如果在此应用程序中设置断点，要确保其在文件复制到远程调试计算机之前设置好。两



台计算机上的文件必须相同。

错误监视

如果接收到有关缺少 DLL 或需要符号信息等方面的警告，要允许系统尽可能多地解决这类问题。通常，这对于一个成功的调试过程是足够的。

在下一部分，我们将开始调查问题可能在控件代码的什么地方。

15.2.4 查找问题

在前面示出的图 15-14 中，我们看到控件返回日期和时间信息时钟面由圆形变成椭圆形。而钟面应该保持圆形，所以必须调查这一问题。下面是我们从 Control Test Container 中控件的初始运行了解到的工程的一些情况：

- 此 ActiveX 控件大部分工作。
- 钟面初始时呈现正确的形状和颜色。
- 日期和时间报告正确。
- 钟面能够正确地画出。
- 当单击钟面时，颜色、时间和日期是正确的——只有钟面的形状是不正确的。

能够远程调试之后，在该 ActiveX 控件中设置两个断点。一个断点放在方法 OnDraw() 中，在此处钟面能够正确地画出。而另一个断点放在方法 HitClock() 中，问题似乎发生在这里。

图 15-16 显示了当我们对 OnDraw() 进行单步调试时此方法返回的两个变量 rc 和 rcBounds 的值。

rcBounds 中的值描述 ActiveX 控件 Clock 的边界矩形，而变量 rc 中的值描述用来画钟面的椭圆(实际上是圆)的边界矩形。rc 边界矩形的大小总是等于或小于 rcBounds 矩形。描述 rc 边界矩形时，为了形成一个圆形的钟面，rc 总是等于 rcBounds 中两个尺寸(水平和垂直)的较小者。实际上，只有当 rcBounds 矩形为正方形时，这两个矩形才是相同的。

在图 15-17 中，rc 边界矩形的值过大。错误明显出在此处。

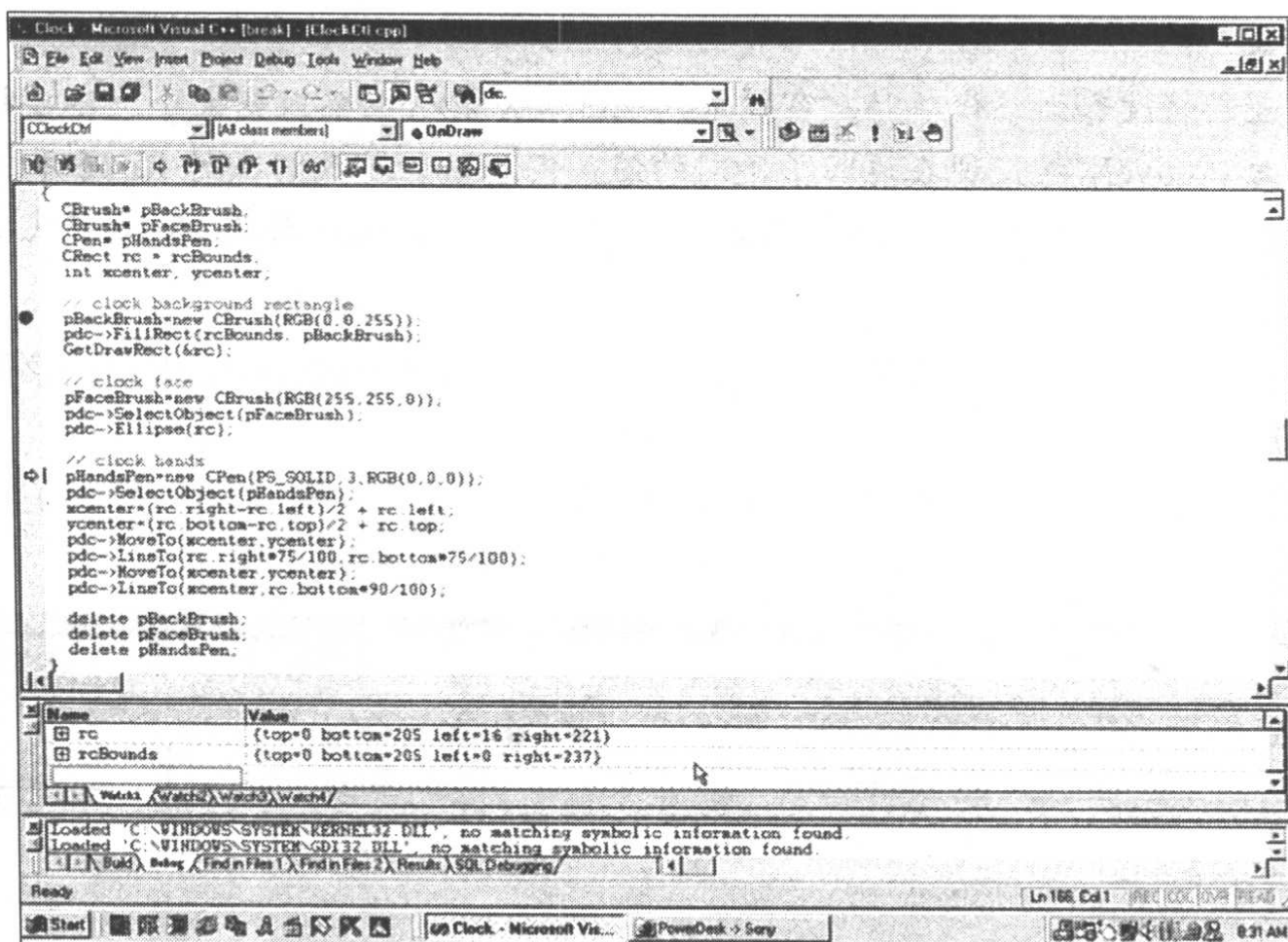


图 15-16 两个变量的值描述了正确画出的钟面

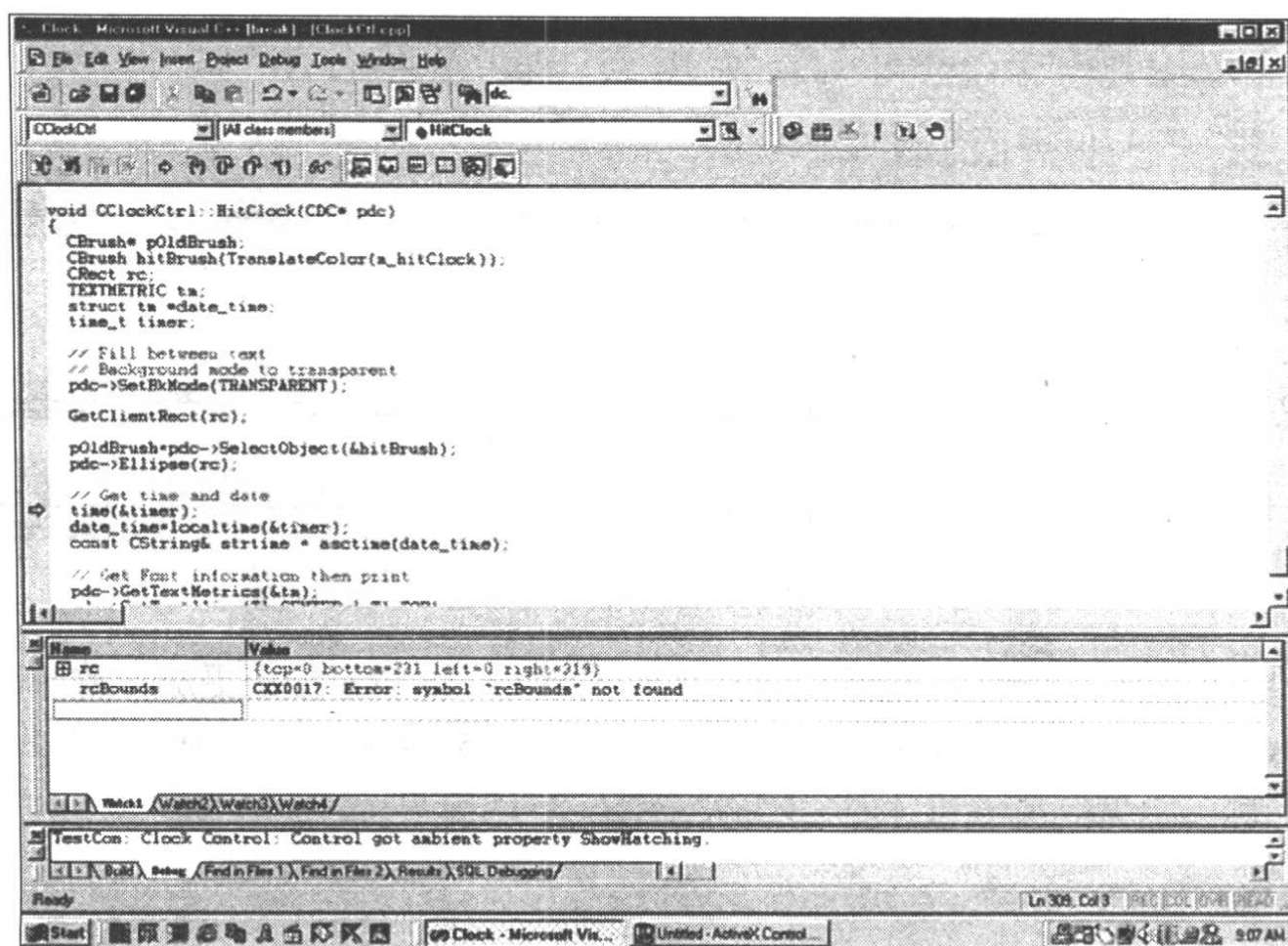


图 15-17 在 HitClock()中检查钟面的边界矩形



错误监视

在我们来回修改和调整代码时，窗口中的 ActiveX 控件必然改变大小。不要被此处的变化误导，以为 ActiveX 控件现在返回了不同的值。例如，如果我们在图 15-16 和图 15-17 之间很好地调整了 ActiveX 控件的大小，则图 15-17 中报告的 rc 值将与图 15-16 中变量 rcBounds 的值一致。

现在，我们已经发现的是变量 rc 中返回的值不是由变量 rcBounds 正确调整得来的值，而是变量 rcBouns 本身。

为改正这一错误，需要使用一个对 GetDrawRect() 的调用。图 15-18 显示了将此函数调用插入到方法 HitClock() 的何处。

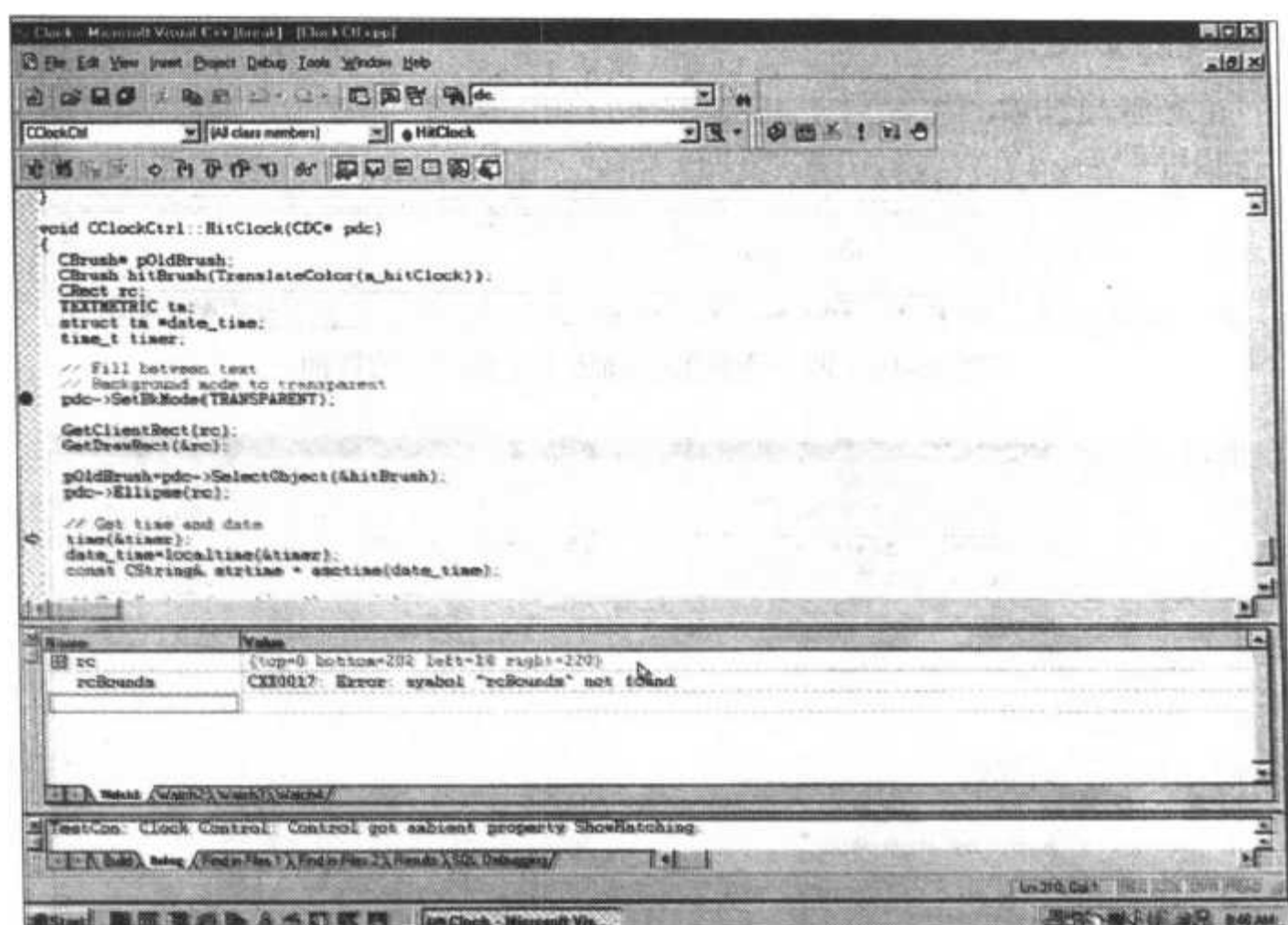


图 15-18 将 GetDrawRect() 插入到方法 HitClock() 中

然后单步调试 HitClock()，显示变量 rc 的值与图 15-16 所示的值更加一致。在 Test Container 中再次测试控件，单击钟面，图 15-19 显示了正确的结果。

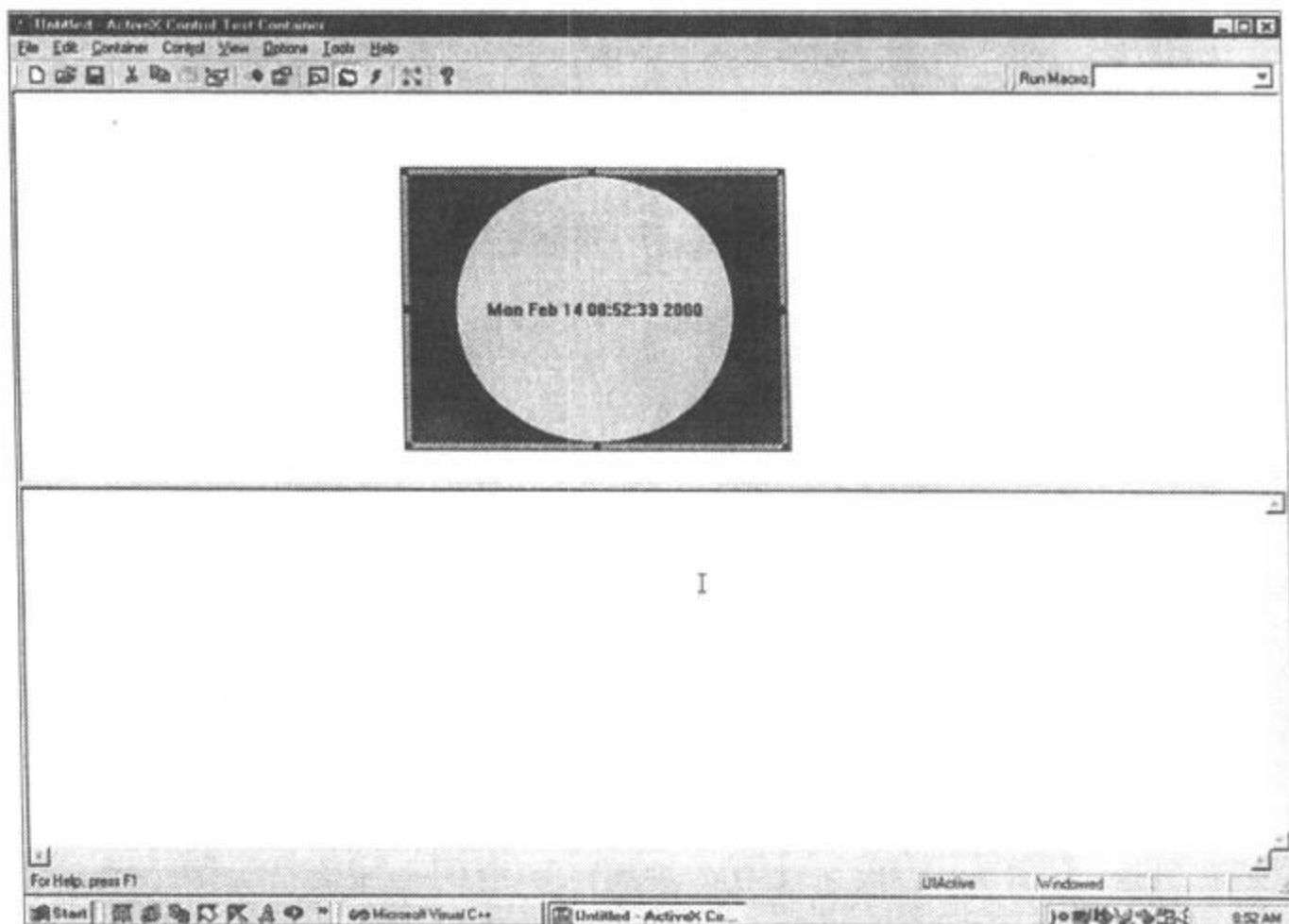


图 15-19 现在 Clock 控件的钟面正确地画出来了

这一应用程序中的代码问题不是什么大的挑战，但是我相信许多读者将同意，设置两台计算机调试一个 ActiveX 控件比前面的许多例子有趣多了。

15.3 小结

本章分为两个主要部分，一部分讲解开发一个 ActiveX 控件的基础，另一部分介绍在 Control Test Container 中使用两台计算机和远程调试技术测试控件的基本步骤。

我们的目的是，使 ActiveX 控件的编码问题尽量简单，以集中讲解在使用两台计算机上调试控件的情况下，插入 Control Test Container 所需要的较为复杂的设置。本章介绍的技术可以很容易地应用到更加可靠的 ActiveX 控件的调试过程中。 ■

第 16 章

调试 COM、ATL 和 DHTML





在前面一章，学习了如何为 ActiveX 控件使用调试工具。本章将进一步讨论如何建立基本的 ActiveX 控件和如何确定调试进程。

和前面一章那样，本章也分为两个部分。第一部分致力于用 ATL COM AppWizard 开发一个简单而实用的工程。在这一部分，将学习建立模板工程所需要的步骤，第二部分将展示如何调试 ATL COM 工程。

16.1 COM 对象模型

ActiveX 控件实际上建立在 COM “对象模型”上，COM 对象模型定义了一个对象如何暴露自身以及这种暴露如何跨过程工作。当我们学会添加另一种强有力的组件——ATL 的功能时，我们使用 COM 对象模型编程的能力将进一步提高。ATL，即 Active Template Library(活动模板库)可很容易地用来产生多种 COM 对象，包括 ActiveX 控件在内。ATL 也为许多基本 COM 接口提供了内部支持。

在接下来的几节中，我们将用 ATL COM AppWizard 开发一个简单的 ATL 应用程序。为了提高效率，我们使用 Microsoft Polygon ATL 教学程序和前面一章开发的 ActiveX 控件的功能开发一个应用程序。我们开发的 COM 对象可以用在 DHTML 文件中，这个文件可以在 Microsoft 的 Internet Explorer 中查看。

16.2 创建一个 ATL 多边形工程

ATL 工程是用 ATL COM AppWizard 产生的，本节开发的工程与 Microsoft 的 Polygon 教学模型紧密相关。笔者强烈建议，在开发这一部分所用的 ATL COM 控件时，要遵循 Microsoft 指南。

下面几步概述了创建基本的 ATL Polygon 工程所需步骤：

1. 从 Visual C++ 中，选择 File | New 菜单选项，然后选择 Projects 标签。
2. 选择 ATL COM AppWizard。
3. 为工程命名为 Polygon，如图 16-1 所示。
4. 单击 OK 按钮后，ATL COM AppWizard 对话框将打开，如图 16-2 所示。
5. 选择 Dynamic Link Library(DLL)选项，然后单击 Finish 就完成了。New Project Information 对话框给出了有关该 ATL 工程的信息，如图 16-3 所示。
6. 单击 OK 按钮，生成 ATL Polygon 工程的基本文件。
7. 一个控件将被添加到基本代码中，以使工程具有功能。使用 Insert | New ATL Object 菜单选项打开 ATL Object Wizard 对话框，如图 16-4 所示。
8. 选择 Full Control 选项，然后单击 Next 按钮。这时可以使用 ATL Object Wizard Properties

对话框的属性页为控件设定各种配置，如图 16-5 所示。

9. 在 Names 标签的 Short Name 文本框内输入名称“PolyCtl”，所有其他输入将自动完成，仍如图 16-5 所示。

10. 下一步选择 Attributes 标签，选择两个 support(支持)复选项，如图 16-6 所示。

11. 现在选择 Stock Properties 标签后使对 Fill Color 的支持有效，如图 16-7 所示。

12. 单击 OK 按钮，回到 Debugger 的屏幕。如果现在打开 File View 窗口，将看到为此工程生成的文件列表，如图 16-8 所示。

13. 现在可以用 AppWizard 生成的模板代码建立 ATL COM 工程。从菜单上选择 Build | Rebuild All 选项编译并链接工程。

14. 和前面一章的 ActiveX 控件那样，可以使用 Control Test Container，在任何点测试这个工程。选择 Tools | ActiveX Control Test Container 菜单选项测试初始控件。

控件实际上暂时还不实用，我们还必须为这一控件添加许多属性、事件和属性页。

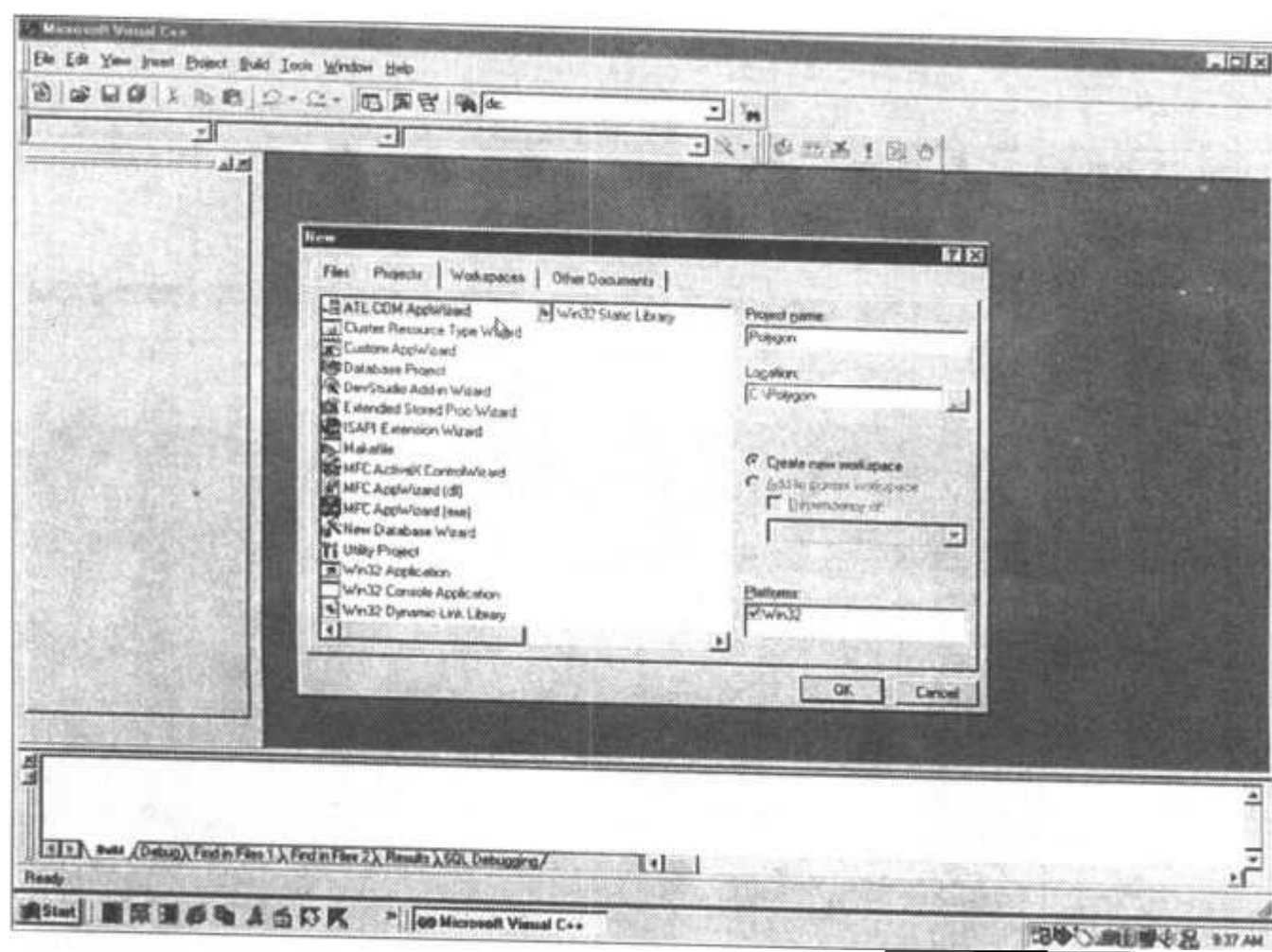


图 16-1 为 ATL Polygon 工程使用 New 对话框

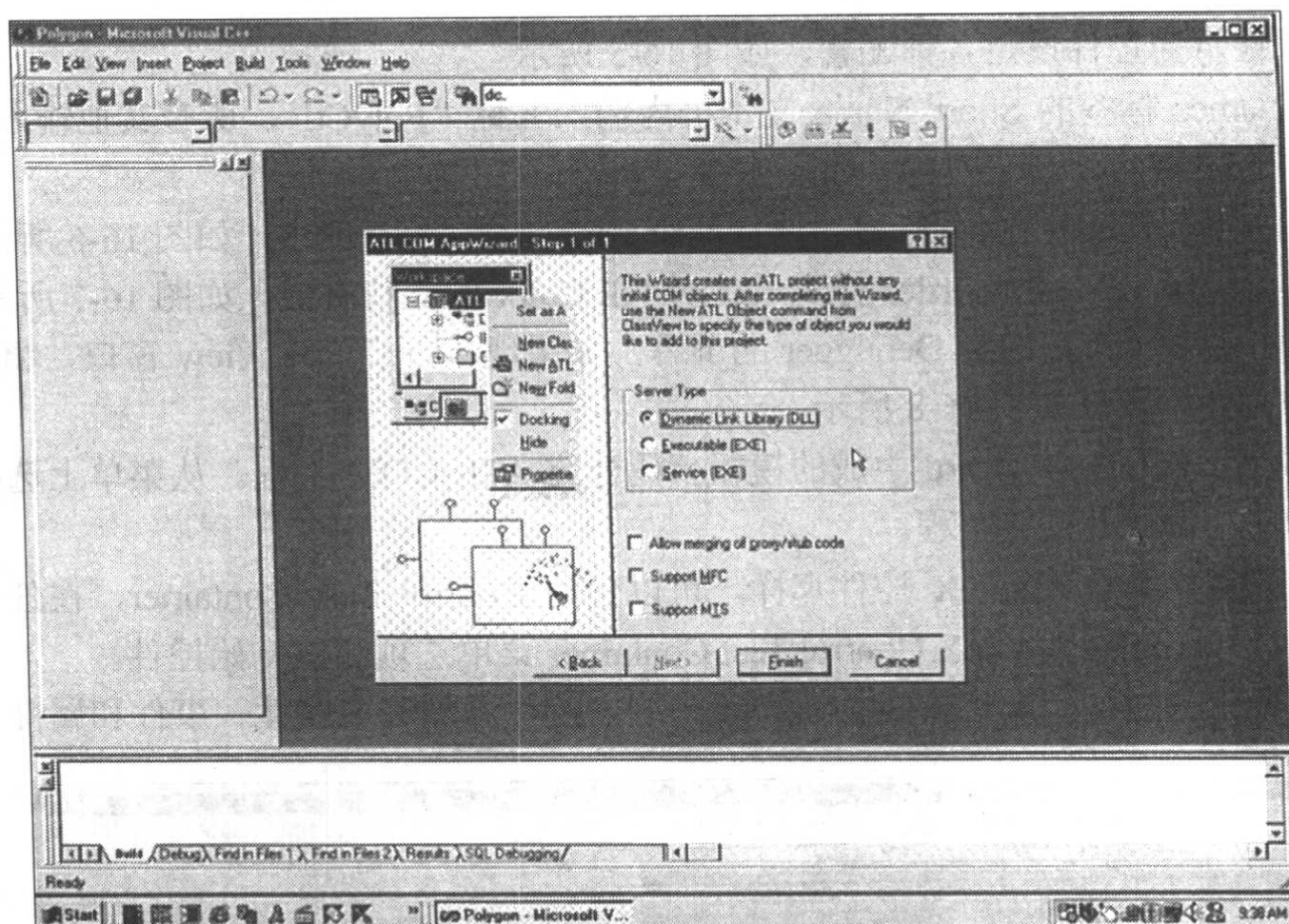


图 16-2 检查 ATL COM AppWizard 对话框

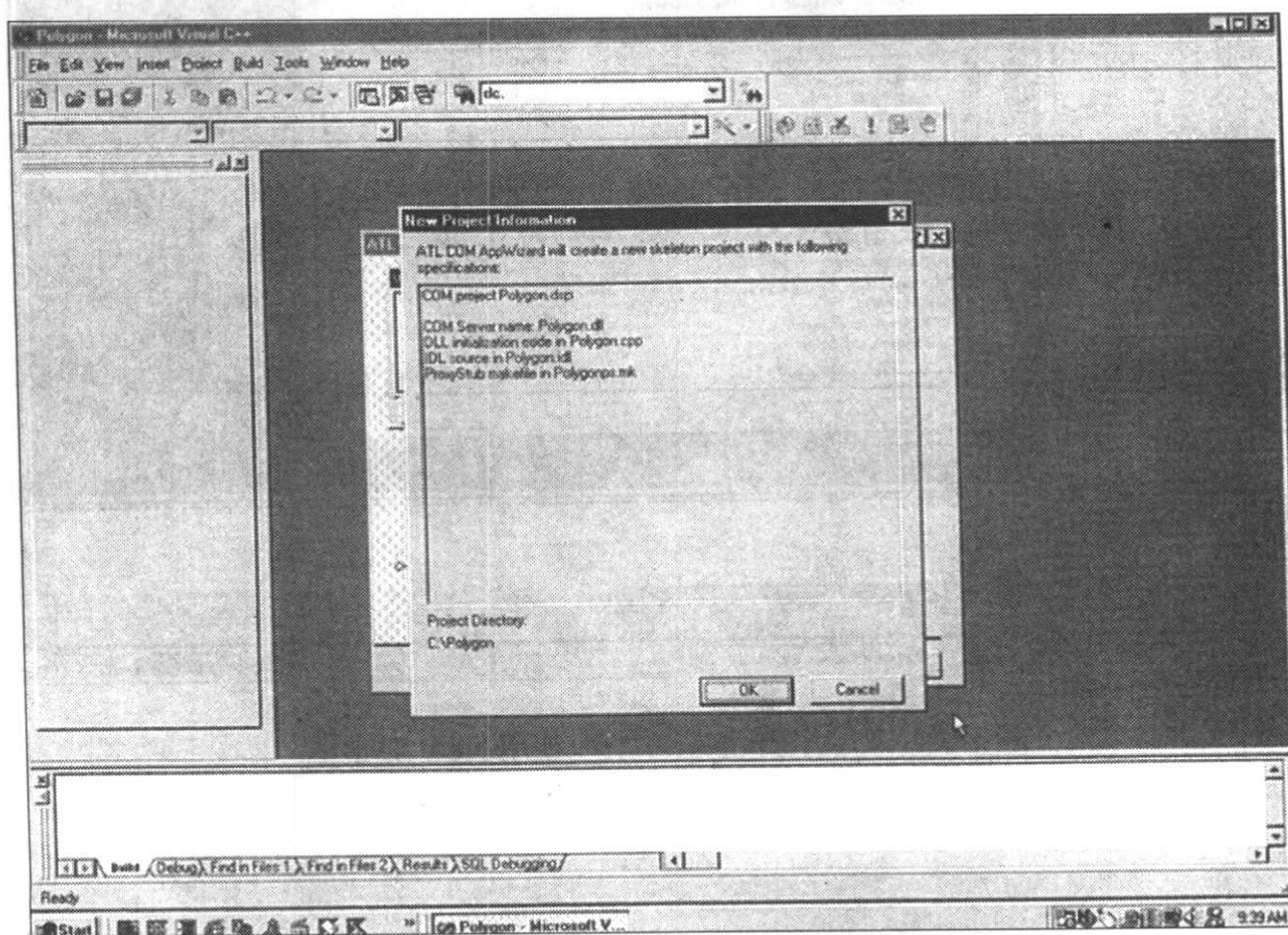


图 16-3 使用 New Project Information 对话框

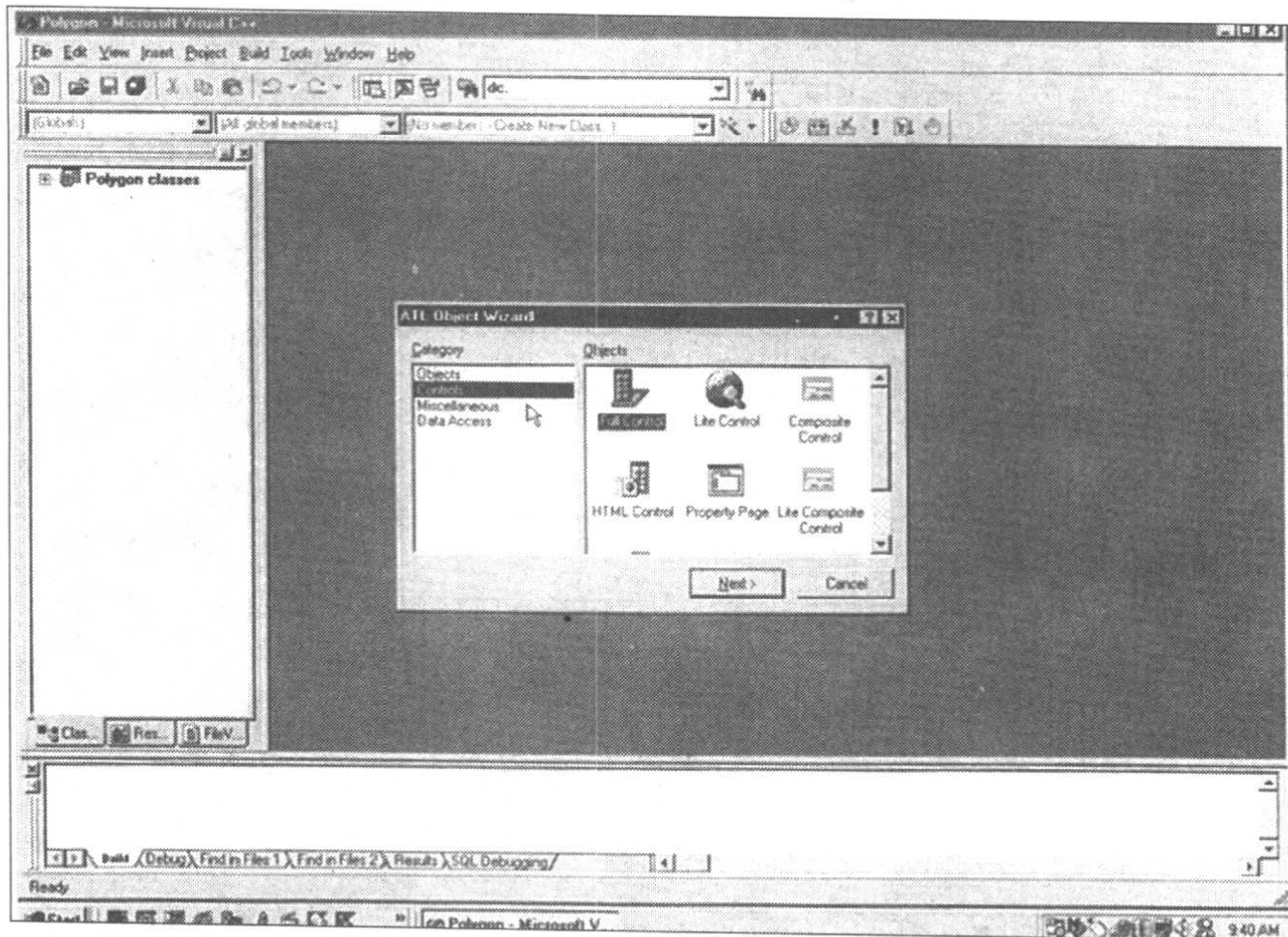


图 16-4 可以使用 ATL Object Wizard 对话框添加一个控件到工程中

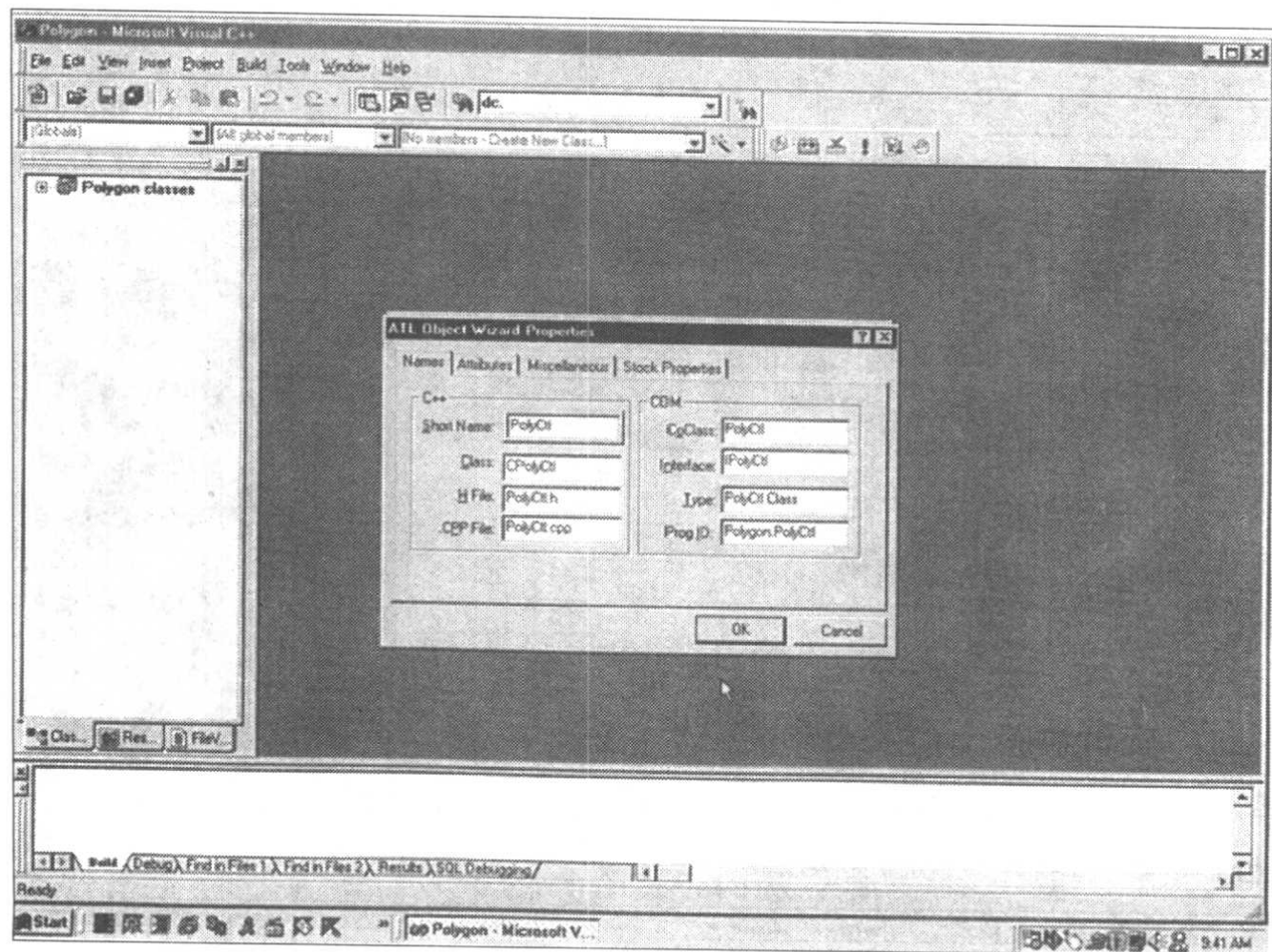


图 16-5 在 ATL Object Properties 对话框中设置控件配置

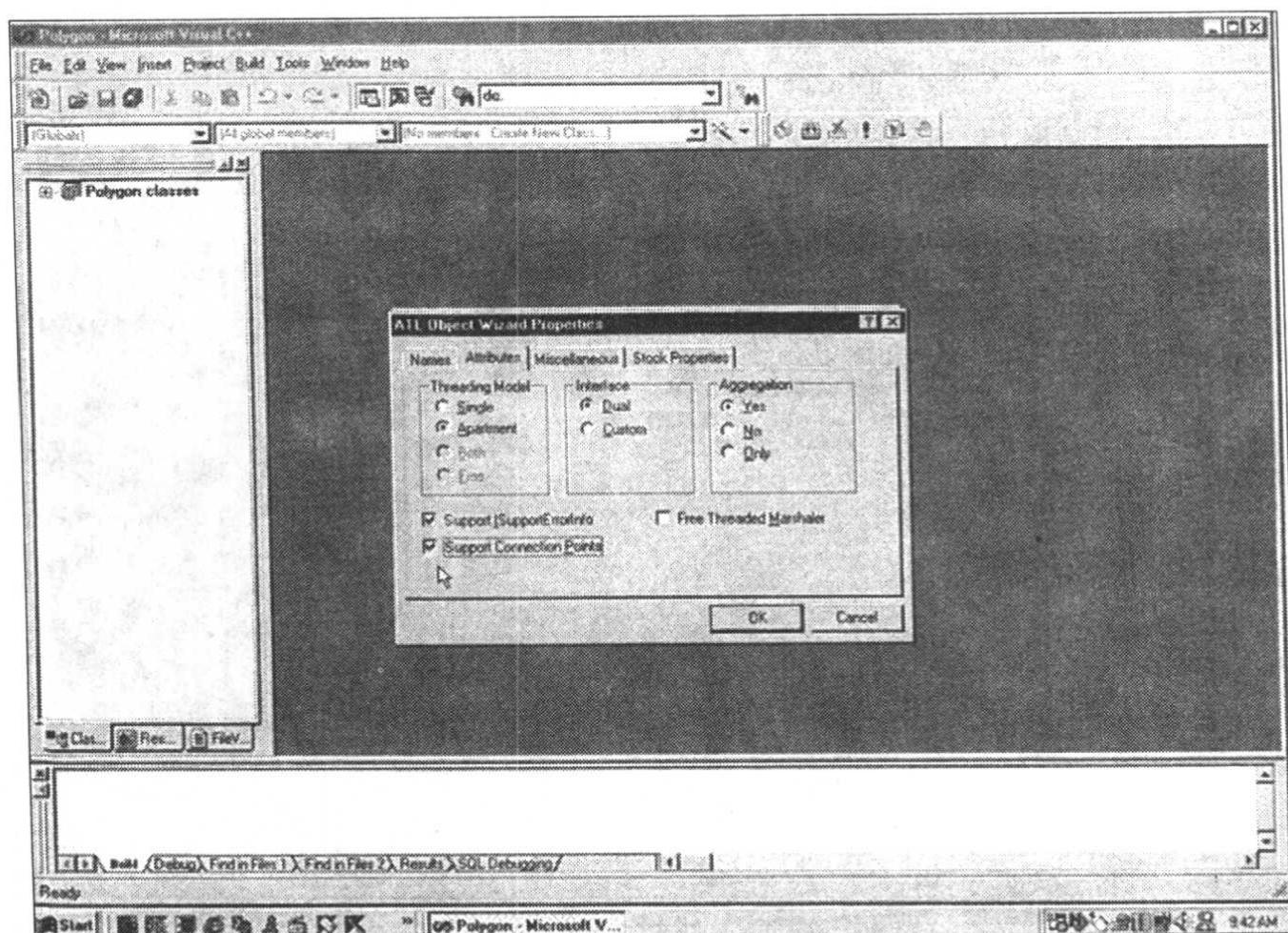


图 16-6 使用 Attributes 标签选择 Support ISupportErrorInfo 和 Support Connection Points

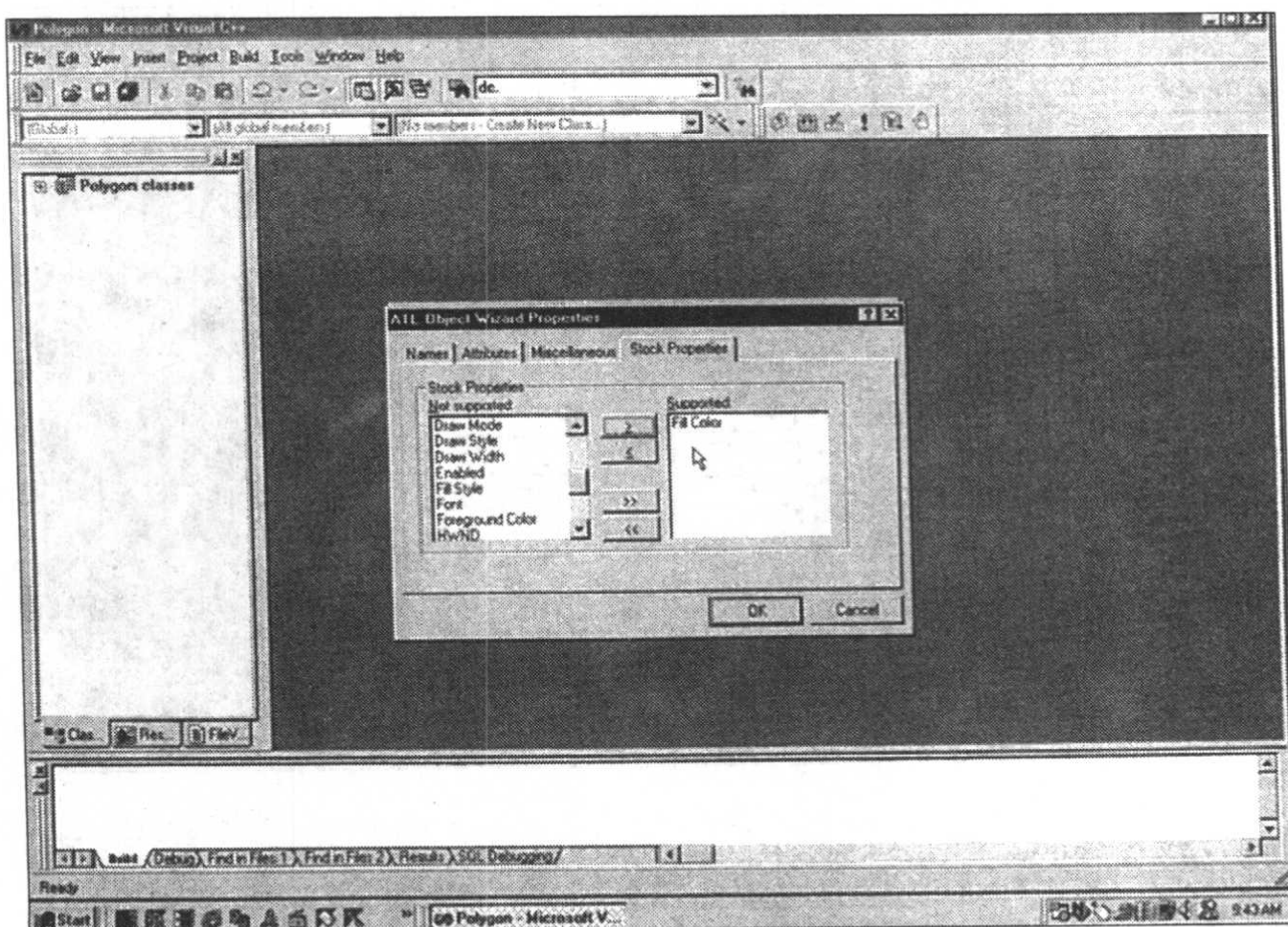


图 16-7 使用 Stock Properties 标签为工程设置特定 stock 属性

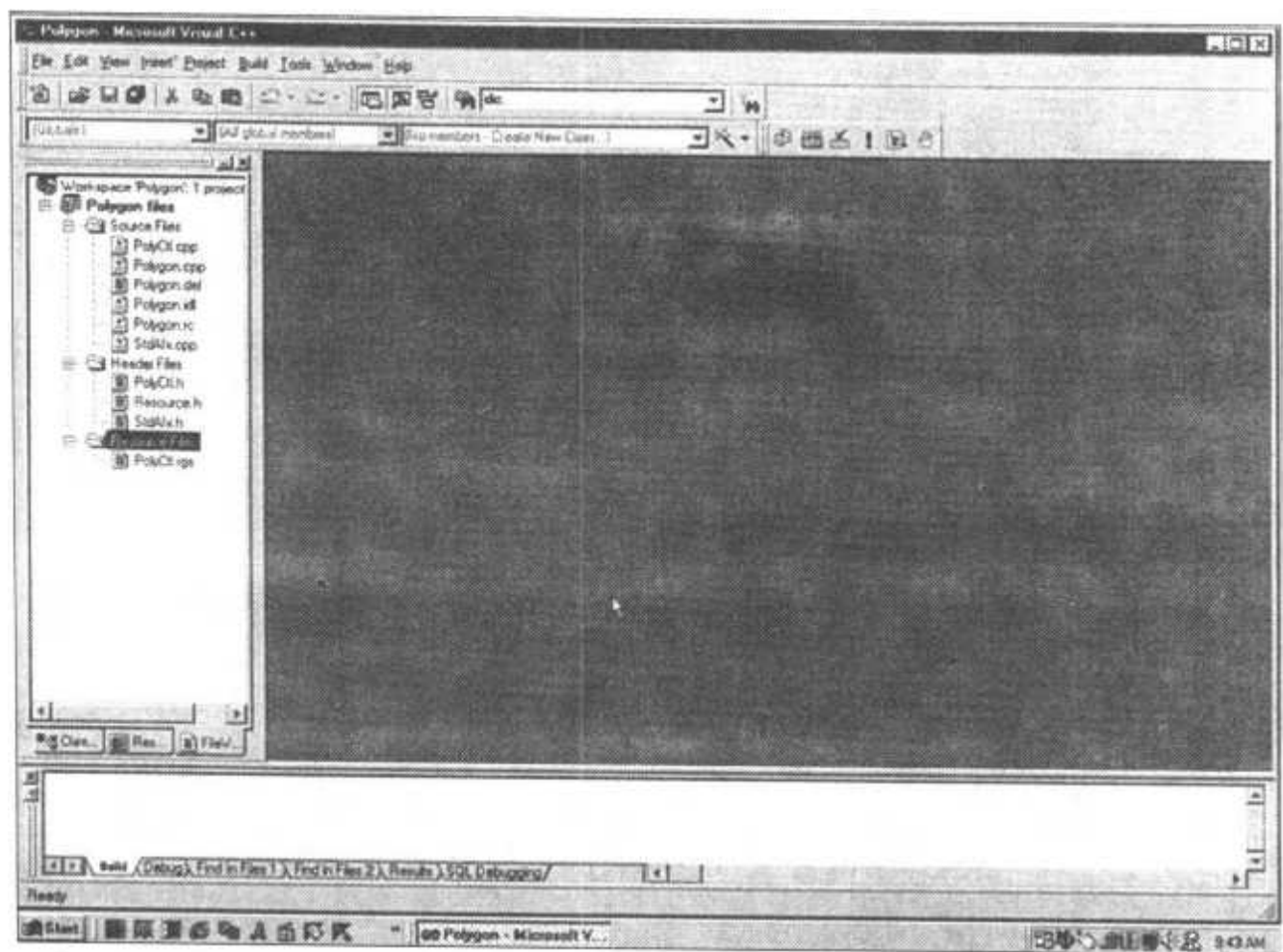


图 16-8 在 FileView 窗口中显示了工程文件列表

16.2.1 优化模块代码

为了使控件实用，我们将修改工程 ATL Polygon 的各种文件。

16.2.1.1 控件的属性

IPolyCtl 用来包含工程的自定义方法和属性。为添加一种新的属性，在 Visual C++ 的 ClassView 窗口中，选中 IpolyCtl 类，然后右击打开一个快捷菜单。在菜单中，选择 Add Property 选项，将打开 Add Property to Interface 对话框，如图 16-9 所示。

从属性类型下拉列表中选择属性类型为短整型(short)，然后以“Sides”为属性名称(Property Name)。单击 OK 按钮，添加新属性。

MIDL(一个建立以.idl 为扩展名的文件的程序)定义了一种 get 方法和一种 put 方法用来检索和设置 Sides 属性。

另外，get 和 put 函数原型被添加到头文件 PolyCtl.h 中，而两者的框架实现被添加到文件 PolyCtl.cpp 中。在下一部分，当查看完整的文件时，将看到这一实现。

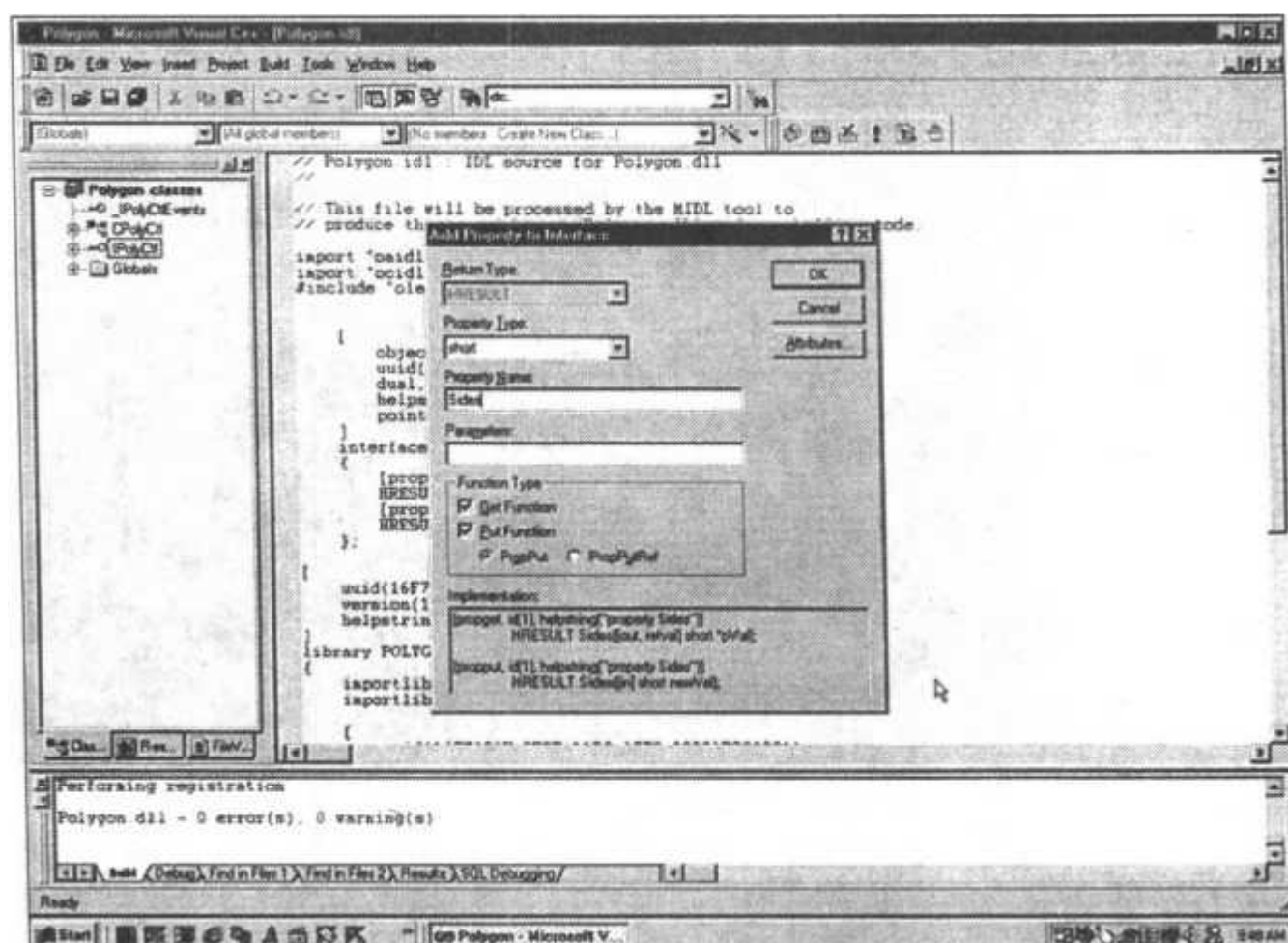


图 16-9 使用 Add Property to Interface 对话框，可以添加 Sides 属性到控件中

16.2.1.2 检查基本的控件代码

工程的模板代码中已经做了许多改动。其中大多数是在文件 PolyCtl.h 和 PolyCtl.cpp 中完成的。

此处是头文件 PolyCtl.h 的代码。添加和修改的基本代码以黑体字显示：

```
// PolyCtl.h : Declaration of the CPolyCtl
#ifndef __POLYCTL_H_
#define __POLYCTL_H_
#include <math.h>
#include "resource.h" // main symbols
#include <atlctl.h>
#include "PolygonCP.h"
////////////////////////////////////
// CPolyCtl
class ATL_NO_VTABLE CPolyCtl :
public CComObjectRootEx<CComSingleThreadModel>,
public CStockPropImpl<CPolyCtl, IPolyCtl, &IID_IPolyCtl, &LIBID_POLYGONLib>,
public CComControl<CPolyCtl>,
public IPersistStreamInitImpl<CPolyCtl>,

```

```

public IOleControlImpl<CPolyCtl>,
public IOleObjectImpl<CPolyCtl>,
public IOleInPlaceActiveObjectImpl<CPolyCtl>,
public IViewObjectExImpl<CPolyCtl>,
public IOleInPlaceObjectWindowlessImpl<CPolyCtl>,
public ISupportErrorInfo,
public IConnectionPointContainerImpl<CPolyCtl>,
public IPersistStorageImpl<CPolyCtl>,
public ISpecifyPropertyPagesImpl<CPolyCtl>,
public IQuickActivateImpl<CPolyCtl>,
public IDataObjectImpl<CPolyCtl>,
public IProvideClassInfo2Impl<CLSID_PolyCtl, &IID_IPolyCtlEvents, &LIBID_POLYGONLib>,
public IPropertyNotifySinkCP<CPolyCtl>,
public CComCoClass<CPolyCtl, &CLSID_PolyCtl>,
public CProxy_IPolyCtlEvents< CPolyCtl >
{
public:
    CPolyCtl()
    {
        m_nSides = 4;                //initial rectangle
        m_clrFillColor = RGB(0xFF, 0xFF, 0); //use yellow fill
    }
DECLARE_REGISTRY_RESOURCEID(IDR_POLYCTL)
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP(CPolyCtl)
    COM_INTERFACE_ENTRY(IPolyCtl)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(IViewObjectEx)
    COM_INTERFACE_ENTRY(IViewObject2)
    COM_INTERFACE_ENTRY(IViewObject)
    COM_INTERFACE_ENTRY(IOleInPlaceObjectWindowless)
    COM_INTERFACE_ENTRY(IOleInPlaceObject)
    COM_INTERFACE_ENTRY2(IOleWindow, IOleInPlaceObjectWindowless)
    COM_INTERFACE_ENTRY(IOleInPlaceActiveObject)
    COM_INTERFACE_ENTRY(IOleControl)
    COM_INTERFACE_ENTRY(IOleObject)
    COM_INTERFACE_ENTRY(IPersistStreamInit)
    COM_INTERFACE_ENTRY2(IPersist, IPersistStreamInit)
    COM_INTERFACE_ENTRY(ISupportErrorInfo)
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
    COM_INTERFACE_ENTRY(ISpecifyPropertyPages)
    COM_INTERFACE_ENTRY(IQuickActivate)
    COM_INTERFACE_ENTRY(IPersistStorage)

```



```
COM_INTERFACE_ENTRY(IDataObject)
COM_INTERFACE_ENTRY(IProvideClassInfo)
COM_INTERFACE_ENTRY(IProvideClassInfo2)
COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)
END_COM_MAP()
BEGIN_PROP_MAP(CPolyCtl)
    PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
    PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
    PROP_ENTRY("FillColor", DISPID_FILLCOLOR, CLSID_StockColorPage)
    PROP_ENTRY("Sides", 1, CLSID_PolyProp)
    // Example entries
    // PROP_ENTRY("Property Description", dispid, clsid)
    // PROP_PAGE(CLSID_StockColorPage)
END_PROP_MAP()
BEGIN_CONNECTION_POINT_MAP(CPolyCtl)
    CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
    CONNECTION_POINT_ENTRY(DIID__IPolyCtlEvents)
END_CONNECTION_POINT_MAP()
BEGIN_MSG_MAP(CPolyCtl)
    CHAIN_MSG_MAP(CComControl<CPolyCtl>)
    DEFAULT_REFLECTION_HANDLER()
    MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
END_MSG_MAP()
// Handler prototypes:
//  LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled);
//  LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled);
//  LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);
// ISupportsErrorInfo
    STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid)
    {
        static const IID* arr[] =
        {
            &IID_IPolyCtl,
        };
        for (int i=0; i<sizeof(arr)/sizeof(arr[0]); i++)
        {
            if (InlineIsEqualGUID(*arr[i], riid))
                return S_OK;
        }
        return S_FALSE;
    }
}
```

```
// IViewObjectEx
    DECLARE_VIEW_STATUS (VIEWSTATUS_SOLIDBKGND | VIEWSTATUS_OPAQUE)
// IPolyCtl
public:
    STDMETHOD(get_Sides) (/*[out, retval]*/ short *pVal);
    STDMETHOD(put_Sides) (/*[in]*/ short newVal);
    HRESULT OnDraw(ATL_DRAWINFO& di);
    HRESULT OnLButtonDown(UINT uMsg, WPARAM wParam,
                          LPARAM lParam, BOOL& bHandled);

    short m_nSides;
    OLE_COLOR m_clrFillColor;
    Point m_arrPoint[10];
}
#endif // __POLYCTL_H_
```

注意其中列出的 `get_Sides` 函数和 `put_Sides` 函数。还要注意在构造函数中，多边形边数的初值被设为 4，而填充颜色的初值设为黄色。这一工程将允许绘制具有 3 至 10 条边的多边形。

设计提示

在 Microsoft 的 Polygon 教学工程中，边数的上限是 100。就实际而言，对于这个数没有任何限制，其范围可以按照用户的意愿任意设定。

现在，右击 `CPolyCtl`，然后选择 `Implement Connect Point` 菜单选项。文件 `PolygonCP.h` 中有一个类称为 `CProxy_IPolyEvents`，是从 `IConnectionPointImpl` 中派生出的，还有两个方法 `Fire_ClickIn` 和 `Fire_ClickOut`，这两个方法用来触发控件的事件：

`IConnectionPointContainer` 是在被添加到 `PolyCtl.h` 中的 COM 映射时，通过函数 `QueryInterface()` 暴露的。

通过使用一个连接点映射，通知 `IConnectionPointContainer` 有有效的点。下面是文件 `PolyCtl.h` 的一小段：

```
BEGIN_CONNECTION_POINT_MAP(CPolyCtl)
    CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
    CONNECTION_POINT_ENTRY(DIID_IPolyCtlEvents)
END_CONNECTION_POINT_MAP()
```

添加一个 `WM_LBUTTONDOWN` 事件处理程序，以检测用户何时单击。该事件的原型被包含在 `PolyCtl.h` 文件中，而其实现放在文件 `PolyCtl.cpp` 中。为添加这一消息处理程序，使用 `ClassView` 窗口中的 `CPolyCtl` 并选择菜单中的 `Add Windows Message Handler` 选项。从消息列表中，选择 `WM_LBUTTONDOWN`，然后单击 `Add Handler` 按钮。

以下清单显示文件 `PolyCtl.cpp` 包含了工程中所有的添加部分。添加的代码部分以黑体



字显示:

```
// PolyCtl.cpp : Implementation of CPolyCtl
#include "stdafx.h"
#include "Polygon.h"
#include "PolyCtl.h"
#include <time.h>
#include <string.h>
////////////////////////////////////
// CPolyCtl
HRESULT CPolyCtl::OnDraw(ATL_DRAWINFO& di)
{
    struct tm *date_time;
    time_t timer;
    static TEXTMETRIC tm;

    RECT& rc = *(RECT*)&di.prcBounds;
    HDC hdc = di.hdcDraw;
    COLORREF colFore;
    HBRUSH hOldBrush, hBrush;
    HPEN hOldPen, hPen;
    // Translate m_colFore into a COLORREF type
    OleTranslateColor(m_clrFillColor, NULL, &colFore);

    // Create and select the colors to draw the circle
    hPen = (HPEN)GetStockObject(BLACK_PEN);
    hOldPen = (HPEN)SelectObject(hdc, hPen);
    hBrush = (HBRUSH)GetStockObject(WHITE_BRUSH);
    hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
    const double pi = 3.14159265358979;
    POINT ptCenter;
    double dblRadiusx = (rc.right - rc.left) / 2;
    double dblRadiusy = (rc.bottom - rc.top) / 2;
    double dblAngle = 3 * pi / 2; // Start at the top
    double dblDiff = 2 * pi / m_nSides; // Angle of side
    ptCenter.x = (rc.left + rc.right) / 2;
    ptCenter.y = (rc.top + rc.bottom) / 2;

    // Calculate the points for each side
    for (int i = 0; i < m_nSides; i++)
    {
        m_arrPoint[i].x = (long)(dblRadiusx * cos(dblAngle) + ptCenter.x + 0.5);
        m_arrPoint[i].y = (long)(dblRadiusy * sin(dblAngle) + ptCenter.y + 0.5);
    }
}
```



```

        dblAngle += dblDiff;
    }
    Ellipse(hdc, rc.left, rc.top, rc.right, rc.bottom);
    // brush that will be used to fill the polygon
    hBrush = CreateSolidBrush(colFore);
    SelectObject(hdc, hBrush);
    Polygon(hdc, &m_arrPoint[0], m_nSides);
    // Print date and time
    time(&timer);
    date_time=localtime(&timer);
    const char* strtime;

    strtime = asctime(date_time);
    SetBkMode(hdc, TRANSPARENT);
    SetTextAlign(hdc, TA_CENTER | TA_TOP);
    ExtTextOut(hdc, (rc.left + rc.right)/2, (rc.top + rc.bottom - tm.tmHeight)/2,
        ETO_CLIPPED, &rc, strtime, strlen(strtime)-1, NULL);

    // Select old pen and brush
    SelectObject(hdc, hOldPen);
    SelectObject(hdc, hOldBrush);
    DeleteObject(hBrush);
    return S_OK;
}

LRESULT CPolyCtl::OnLButtonDown(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
{
    HRGN hRgn;
    WORD xPos = LOWORD(lParam); // horizontal position of cursor
    WORD yPos = HIWORD(lParam); // vertical position of cursor
    // Create a region from our list of points
    hRgn = CreatePolygonRgn(&m_arrPoint[0], m_nSides, WINDING);

    // If clicked point is in polygon fire the ClickIn
    // event otherwise fire ClickOut event
    if (PtInRegion(hRgn, xPos, yPos))
        Fire_ClickIn(xPos, yPos);
    else
        Fire_ClickOut(xPos, yPos);
    // Delete the region that we created
    DeleteObject(hRgn);
    return 0;
}

```



```
STDMETHODIMP CPolyCtl::get_Sides(short *pVal)
{
    *pVal = m_nSides;
    return S_OK;
}
STDMETHODIMP CPolyCtl::put_Sides(short newVal)
{
    if (newVal > 2 && newVal < 11)
    {
        m_nSides = newVal;
        return S_OK;
    }
    else
        return Error(_T("Must have between 3 and 10 sides"));
}
```

Microsoft 开发最初的 Polygon 教学工程时, 用 `math.h` 中的 `sin()` 和 `cos()` 函数计算多边形的顶点, 查看可否在前面的程序清单中找到这部分代码。我们的工程修改了最初的 Polygon 工程, 将边数的限制由 100 改为 10, 并且包含了在控件内显示本地日期和时间所需要的代码。而且, 每当在多边形内单击时, 日期和时间将更新。这是前面一章中开发的 ActiveX 控件使用的同样的技术。

为了触发事件, 文件 `Polygon.idl` 中必须添加一小部分代码。文件中添加部分以黑体字显示:

```
// Polygon.idl : IDL source for Polygon.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (Polygon.tlb) and marshalling code.
import "oaidl.idl";
import "ocidl.idl";
#include "olectl.h"

[
    object,
    uuid(16F763AD-9F2E-11D3-A7E0-0080AE000001),
    dual,
    helpstring("IPolyCtl Interface"),
    pointer_default(unique)
]
interface IPolyCtl : IDispatch
{
    [propget, id(DISPID_FILLCOLOR)]
```

```

    HRESULT FillColor([in]OLE_COLOR clr);
    [propget, id(DISPID_FILLCOLOR)]
    HRESULT FillColor([out, retval]OLE_COLOR* pclr);
    [propget, id(1), helpstring("property Sides")] HRESULT Sides([out, retval]
short *pVal);
    [propput, id(1), helpstring("property Sides")] HRESULT Sides([in] short
newVal);
};
[
    uuid(16F763A1-9F2E-11D3-A7E0-0080AE000001),
    version(1.0),
    helpstring("Polygon 1.0 Type Library")
]
library POLYGONLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [
        uuid(16F763AF-9F2E-11D3-A7E0-0080AE000001),
        helpstring("_IPolyCtlEvents Interface")
    ]
    dispinterface _IPolyCtlEvents
    {
        properties:
        methods:
        [id(1), helpstring("method ClickIn")] void ClickIn([in] long x, [in] long
y);
        [id(2), helpstring("method ClickOut")] void ClickOut([in] long x, [in]
long y);
    };
    [
        uuid(16F763AE-9F2E-11D3-A7E0-0080AE000001),
        helpstring("PolyCtl Class")
    ]
    coclass PolyCtl
    {
        [default] interface IPolyCtl;
        [default, source] dispinterface _IPolyCtlEvents;
    };
    [
        uuid(7846CD41-9F37-11D3-A7E0-0080AE000001),
        helpstring("PolyProp Class")
    ]
}

```



```
coclass PolyProp
{
    interface IUnknown;
};
```

ClickIn()和 ClickOut()方法使用单击处的 x 和 y 坐标作为参数。

可以用 ATL Object Wizard 在控件中添加一个属性页，用 Insert | New ATL Object 菜单选项打开 ATL Object Wizard 对话框，如图 16-10 所示。

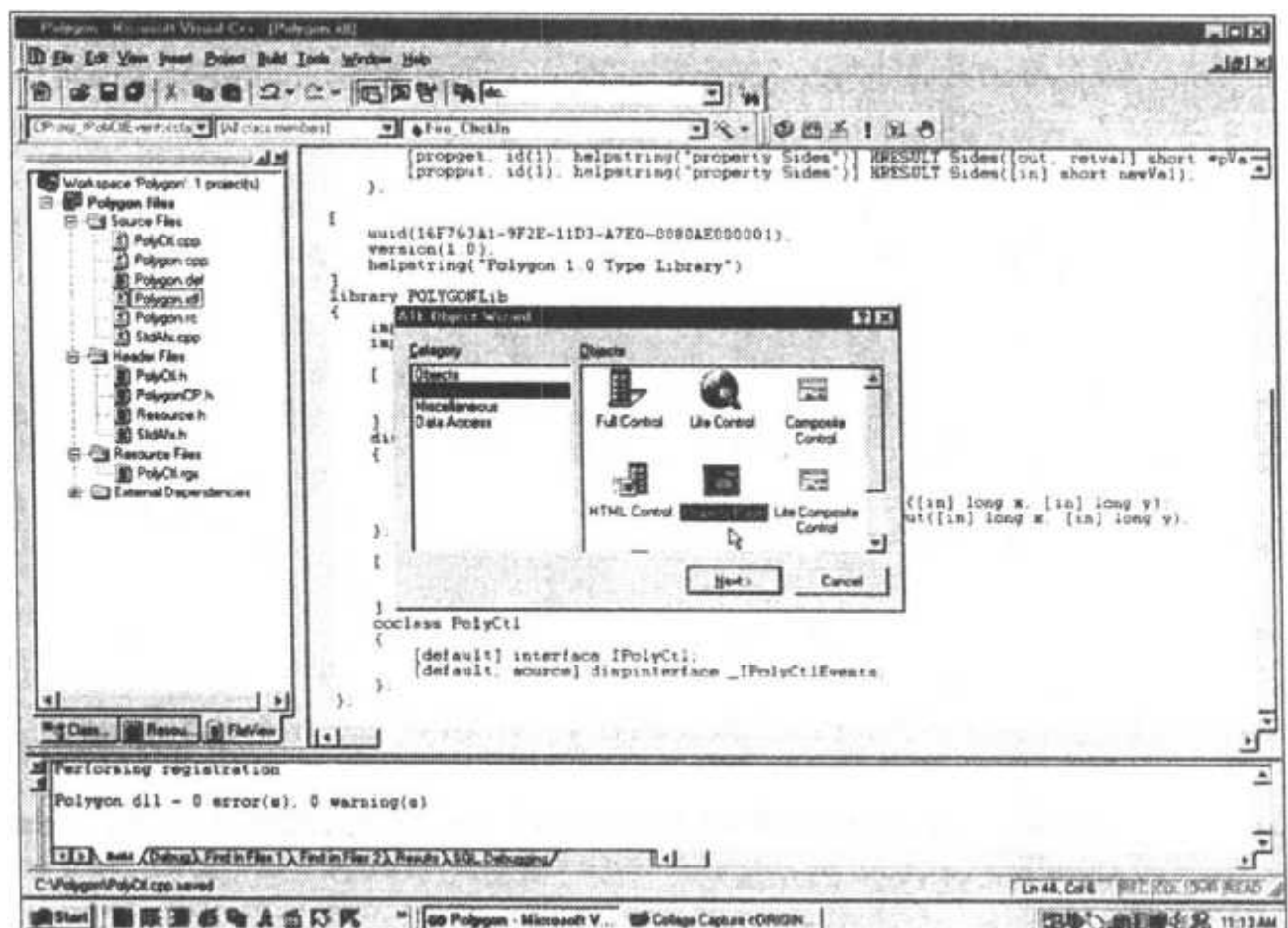


图 16-10 使用 ATL Object Wizard 对话框在工程中添加一个属性页

选择 Property Page 选项，然后单击 Next 按钮。这时允许为使用 ATL Object Wizard 对话框中属性页的控件设定各种配置，如图 16-11 所示。

在此对话框中的 Names 标签的 Short Name 文本框中输入名称“PolyProp”，所有其他输入将自动完成，如图 16-11 所示。

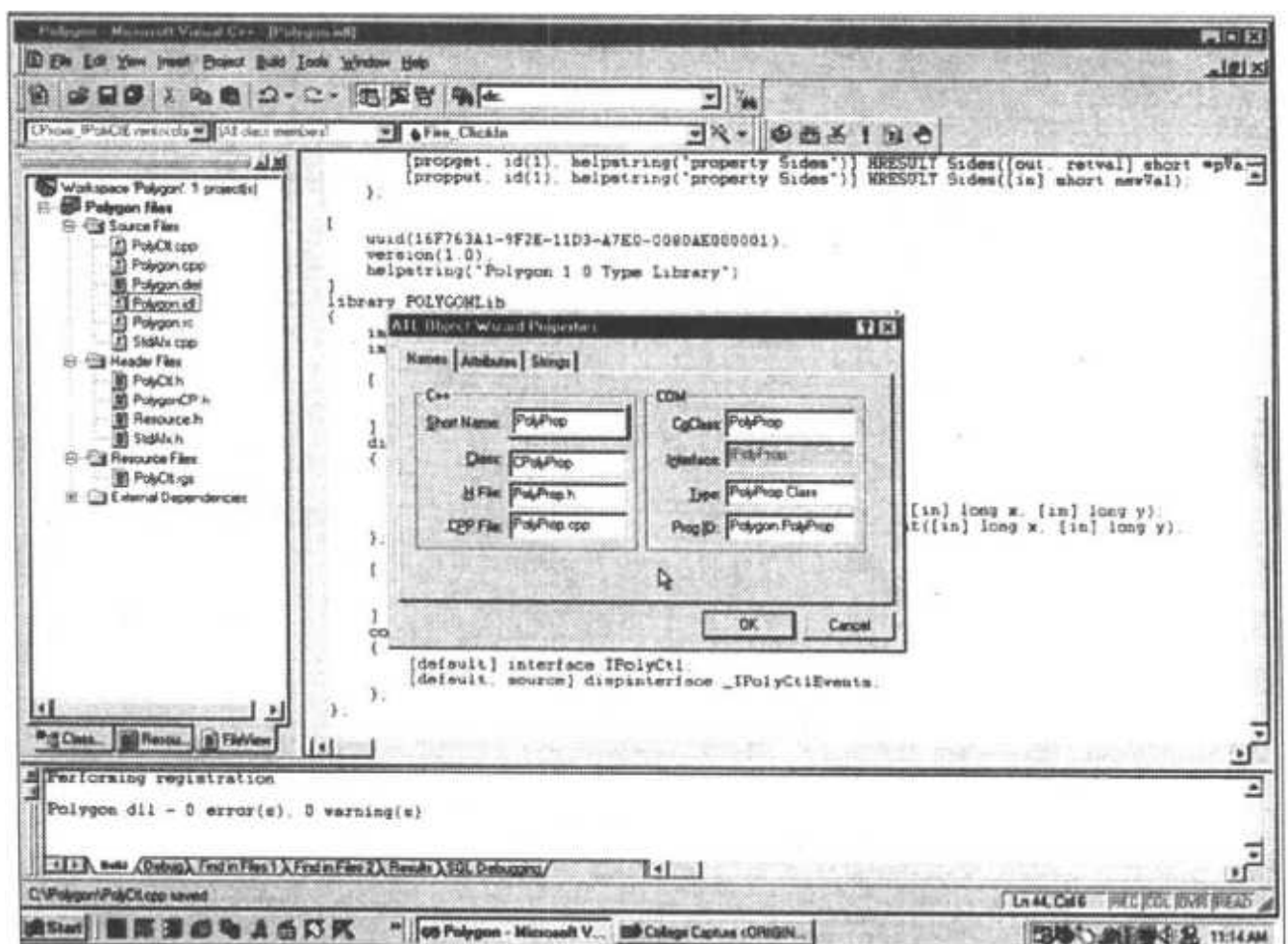


图 16-11 可以使用 ATL Object Wizard Properties 对话框设定各种控件配置

然后选择 Strings 标签，为 Title 和 Doc String 输入值，如图 16-12 所示。

单击 OK 按钮，生成文件 PolyProp.h、PolyProp.cpp 和 PolyProp.rgs。除了这些文件之外，添加一个新的属性页到对象入口映射中。

在 Visual C++ 中使用 ResourceView 窗口打开 IDD_POLYPROP 对话框，将标签改为 Sides，然后添加一个 ID 值为 IDC_SIDES 的文本框控件。

```
// PolyProp.h : Declaration of the CPolyProp
#ifndef __POLYPROP_H_
#define __POLYPROP_H_
#include "resource.h"           // main symbols
#include "Polygon.h"

EXTERN_C const CLSID CLSID_PolyProp;

////////////////////////////////////

// CPolyProp
class ATL_NO_VTABLE CPolyProp :
    public CComObjectRootEx<CComSingleThreadModel>,
```

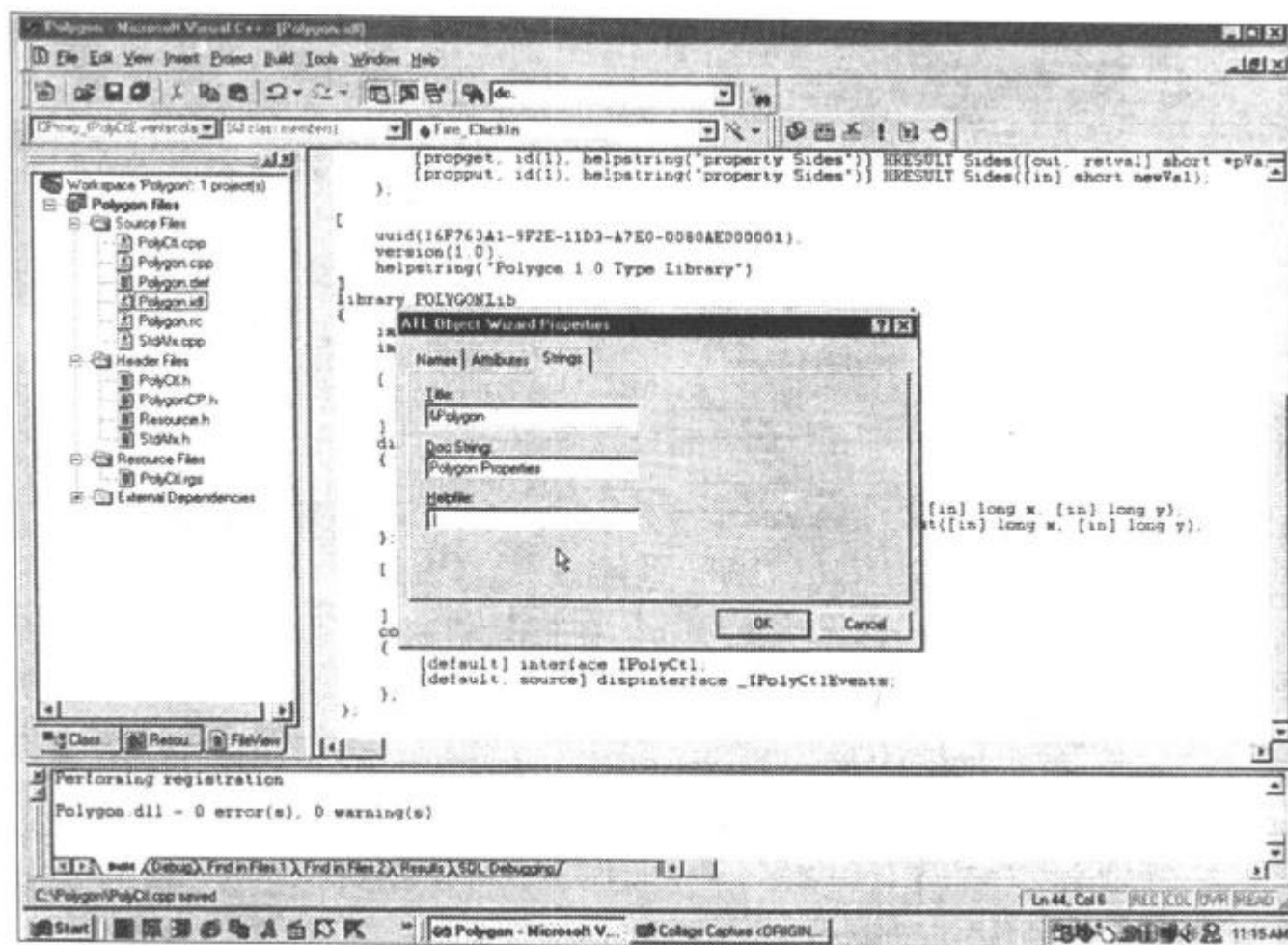


图 16-12 在 Strings 标签中为属性页设置 Title 和 Doc String

```
public CComCoClass<CPolyProp, &CLSID_PolyProp>,
public IPropertyPageImpl<CPolyProp>,
public CDialogImpl<CPolyProp>
{
public:
    CPolyProp()
    {
        m_dwTitleID = IDS_TITLEPolyProp;
        m_dwHelpFileID = IDS_HELPFILEPolyProp;
        m_dwDocStringID = IDS_DOCSTRINGPolyProp;
    }
    enum { IDD = IDD_POLYPROP };
    DECLARE_REGISTRY_RESOURCEID(IDR_POLYPROP)
    DECLARE_PROTECT_FINAL_CONSTRUCT()
    BEGIN_COM_MAP(CPolyProp)
        COM_INTERFACE_ENTRY(IPropertyPage)
    END_COM_MAP()
    BEGIN_MSG_MAP(CPolyProp)
        CHAIN_MSG_MAP(IPropertyPageImpl<CPolyProp>)
    END_MSG_MAP()
}
```

```
// Handler prototypes:
// LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled);
// LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled);
// LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);
STDMETHOD(Apply) (void)
{
    ATLTRACE(_T("CPolyProp::Apply\n"));
    for (UINT i = 0; i < m_nObjects; i++)
    {
        CComQIPtr<IPolyCtl, &IID_IPolyCtl> pPoly(m_ppUnk[i]);
        short nSides = (short)GetDlgItemInt(IDC_SIDES);
        if FAILED(pPoly->put_Sides(nSides))
        {
            CComPtr<IErrorInfo> pError;
            CComBSTR strError;
            GetErrorInfo(0, &pError);
            pError->GetDescription(&strError);
            MessageBox("Error - Enter 3 to 10 sides",
                "Error Message", MB_ICONEXCLAMATION);
            return E_FAIL;
        }
    }
    m_bDirty = FALSE;
    return S_OK;
}
};
#endif // __POLYPROP_H_
```

由于一个属性页可以同时有多个附属于它的客户区，因此使用了函数 Apply() 循环，并用函数 put_Sides 处理每个客户区中的文本框的值。

通过为头文件 PolyCtl.h 添加下面一行，添加属性页到控件：

```
PROP_ENTRY("Sides", 1, CLSID_PolyProp)
```

现在，我们已经准备了在一个实际的 Web 页上测试该控件。

24x7 连接点和 ATL 代理发生器

与前面一章讨论过的 ActiveX 控件不同，这一控件需要一个连接点接口和一个连接点容器接口。由于一个 COM 对象可以拥有多个连接点，所以该 COM 对象需要实现一个连接点容器。接口 IConnectionPoint 用来实现一个连接点容器。

通过读类型库并为每个可以触发的事件产生一个函数，ATL 代理发生器用来产生 IConnectionPoint 接



口。例如，可以在 FileView 窗口中用右击文件 Polygon.idl 为这一工程建立一个类型库。文件 Polygon.tlb 通过选择 Compile Polygon.idl 产生。这一文件成为类型库。

16.2.2 测试控件

ATL Object Wizard 生成了初始控件和一个保存该控件的 HTML 文件。这个文件为 PolyCtl.htm，可以在 Microsoft 的 Internet Explorer 中打开。通过使用 Internet Explorer，将能够在一个实际的 Web 页上查看并测试 ATL 控件。

在 Visual C++ 环境中，以黑体字显示修改的 PolyCtl.htm。这个文件将基于使用的编译器的 ATL 版本而在结构上不同。

```
<HTML>
<HEAD>
<TITLE>ATL 2.0 test page for object PolyCtl</TITLE>
</HEAD>
<BODY>
<OBJECT ID="PolyCtl" <
  CLASSID="CLSID:4CBBC676-507F-11D0-B98B-000000000000">
>
</OBJECT>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub PolyCtl_ClickIn(x, y)
PolyCtl.Sides = PolyCtl.Sides + 1
End Sub
Sub PolyCtl_ClickOut(x, y)
PolyCtl.Sides = PolyCtl.Sides - 1
End Sub
-->
</SCRIPT>
</BODY>
</HTML>
```

现在启动 Internet Explorer，打开文件 PolyCtl.htm，屏幕开始时应如图 16-13 所示。

继续测试控件。在控件的多边形区域内部和外部单击。图 16-14 显示了在控件内部又单击了 4 次后的式样。

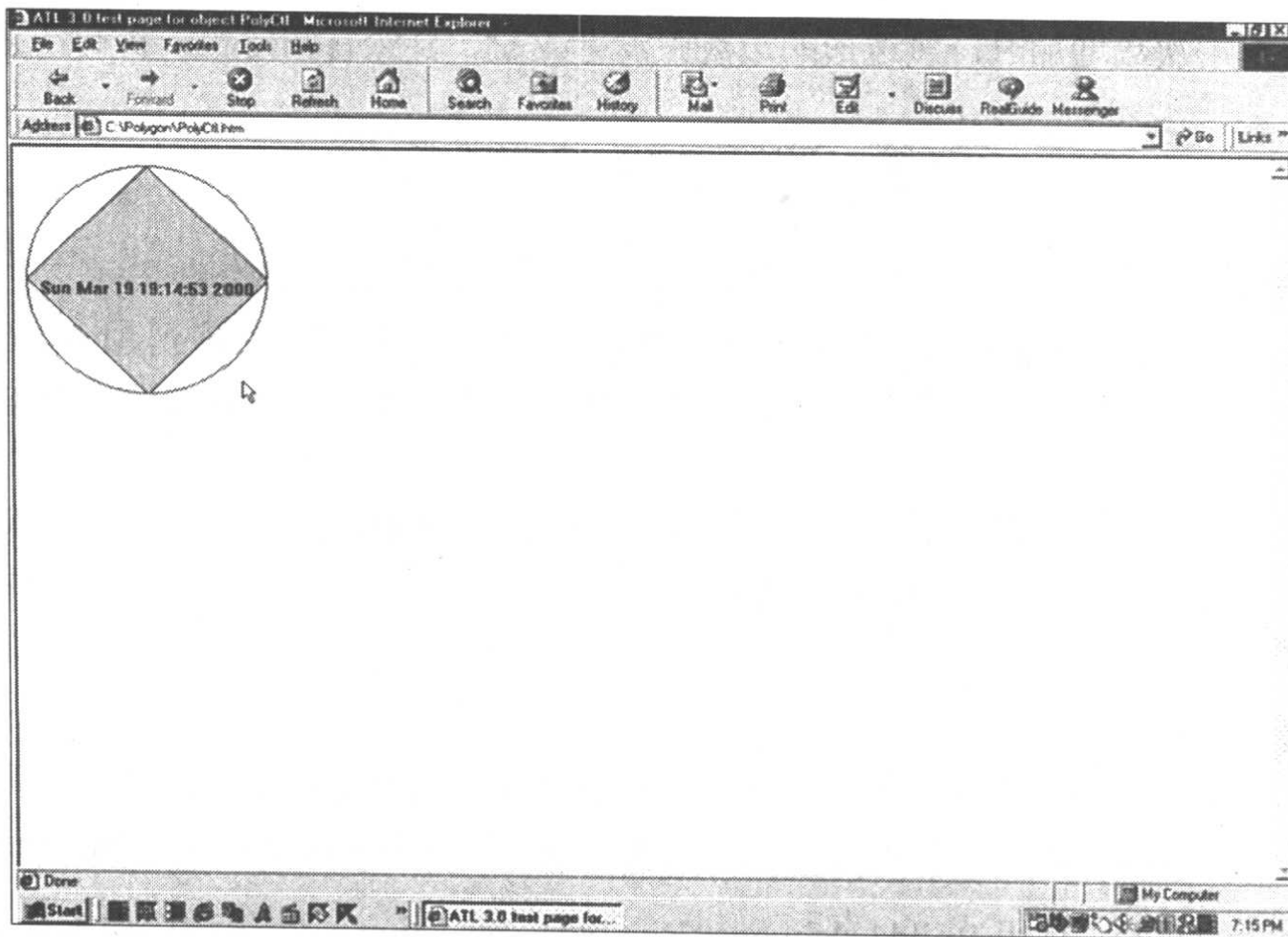


图 16-13 在一个 Web 页上查看初始控件

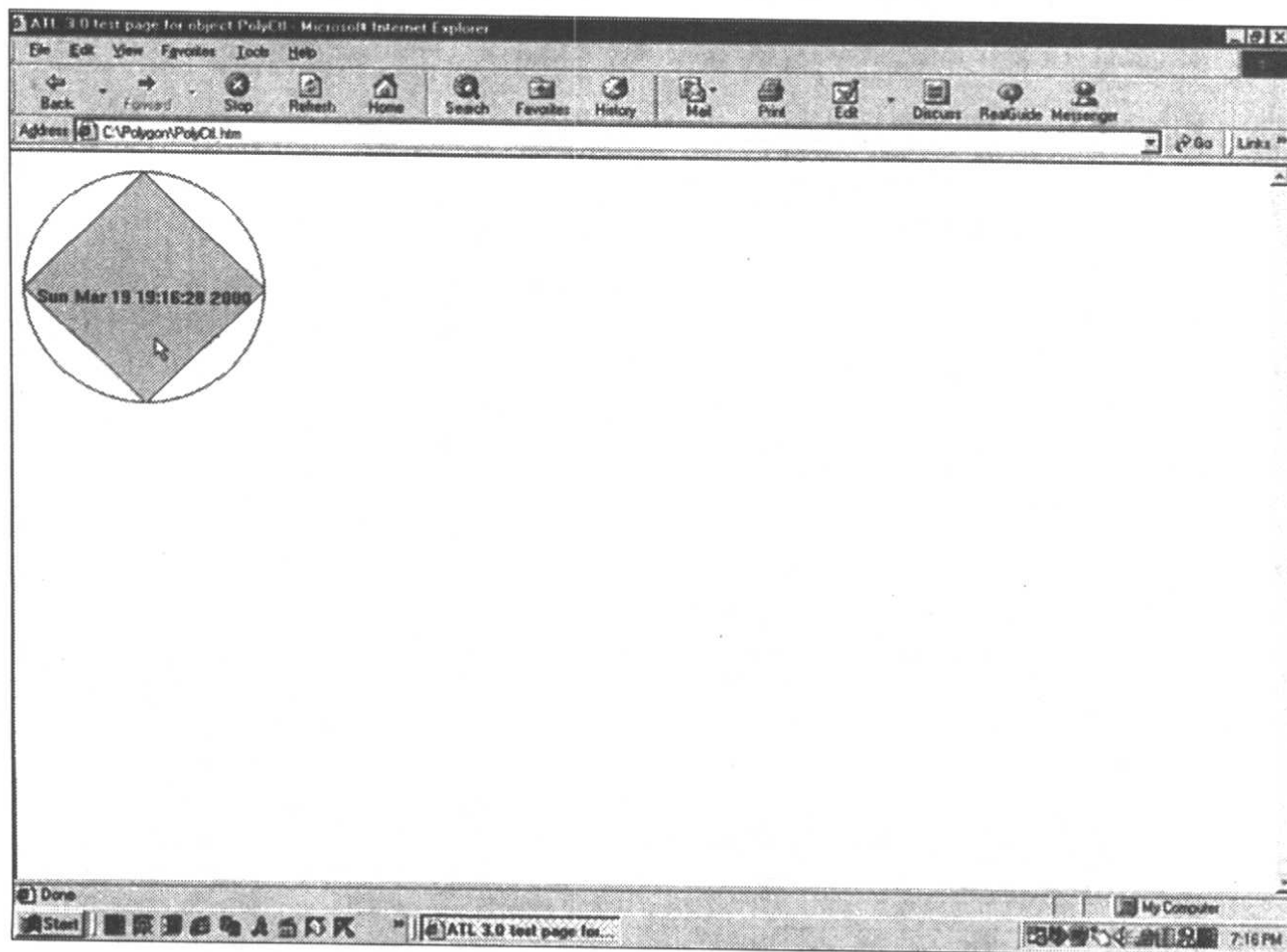


图 16-14 在控件的多边形区域内单击了 4 次



注意当单击次数增加时，多边形的边数并没有改变。显然有些问题，我们不得不调查。

16.3 调试 ATL COM 控件

在开始调试阶段之前，回顾这时我们知道些什么：控件看来部分地工作，它注册了自身，并可以被加载到 Control Test Container 或作为一个 HTML 文档加载到 Microsoft Internet Explorer 中。看来不能正确工作的是“命中”，也就是说在多边形区域内部或单击外部时缺少一个“命中”。

我们需要知道的第一件事情是方法 ClickIn()和 ClickOut()是否正确地工作。如果它们正确工作，成员变量 m_nSides 根据单击的速率改变否？下面进行调试，调查这些问题，这次不使用远程调试设置。

设计提示

和前面一章对 ActiveX 控件做的那样启动一个远程调试阶段是可以的。如果需用远程调试并需要另外的帮助，则可以参考前一章的指导。控件可以在 Control Test Container 中测试。

在本阶段的开始，使用 Build | Debugger Remote Connection 菜单选项启动本地调试过程。图 16-15 显示了在 Watch 窗口中调试变量 m_nSides 的过程。

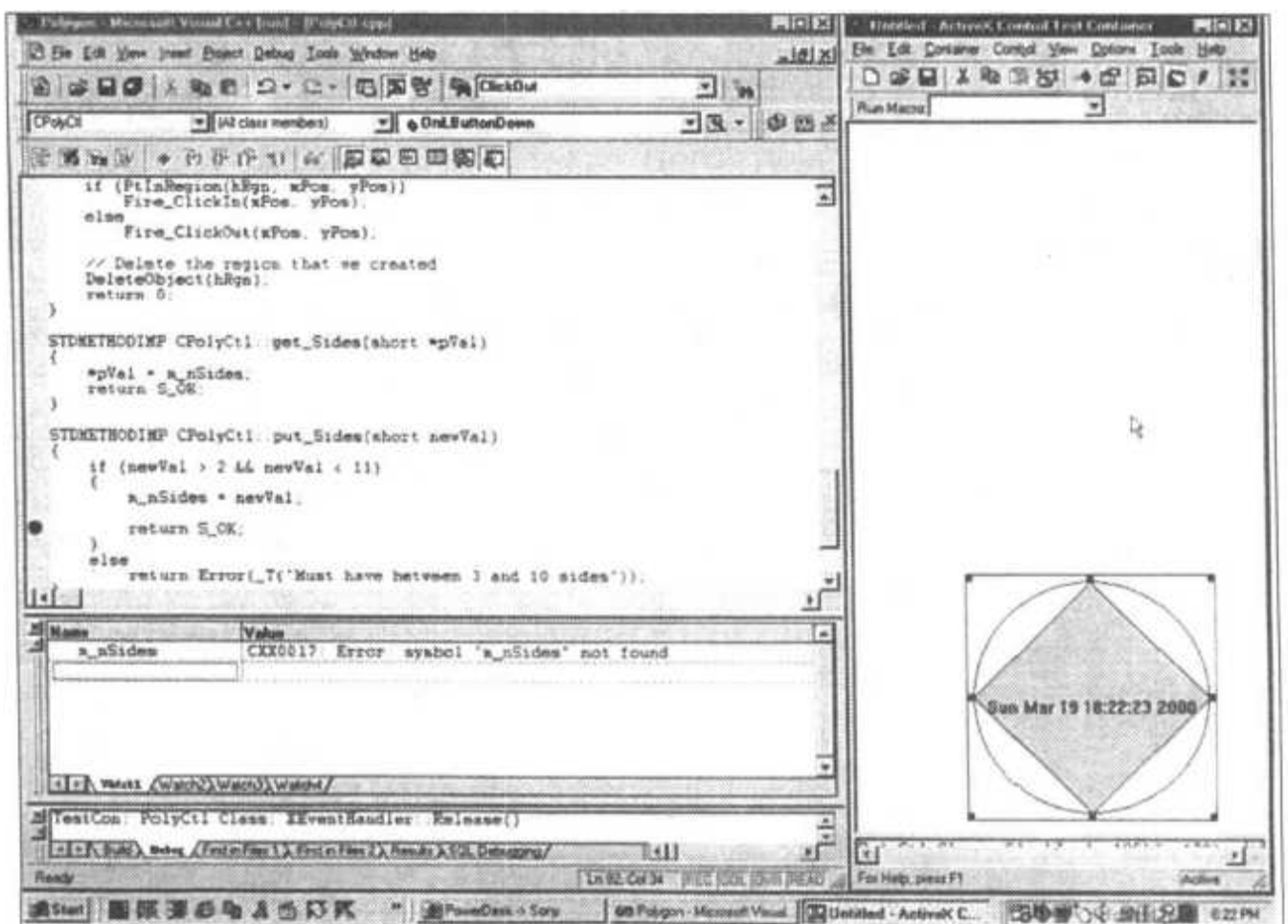


图 16-15 在同一台计算机上查看 Debugger 和 Test Container 窗口

在查看这张图时，还要注意到方法 `put_Sides` 中设置了一个断点。Debugger 窗格在左侧，Control Test Container 窗格在右侧。这时，因为我们还没修改边数，所以变量 `m_nSides` 返回图 16-15 所示的错误信息。

为改变边数，打开 Test Container 中的 Edit 菜单，使用 Property 选项打开并修改控件的边数。图 16-16 显示了边数改为 3 之后的调试情况。

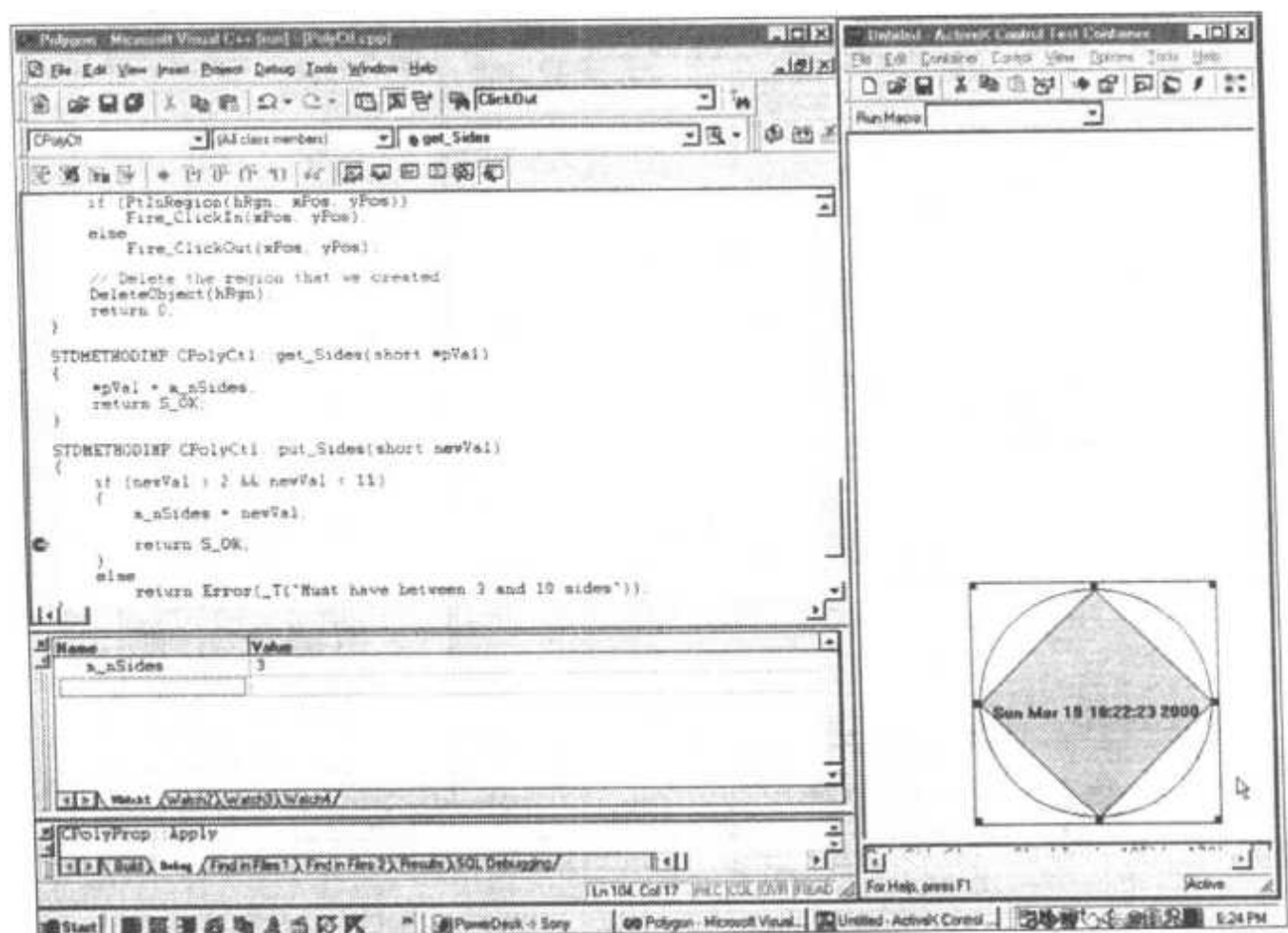


图 16-16 边数已经变为 3

变量 `m_nSides` 反映了这种变化，而图 16-16 中所示的控件仍然是四边形。如果切换到另一个应用程序，然后再切换回 Test Container，将看到控件被正确地刷新。

如果是一个经验丰富的 Windows 程序员，可以看出问题的起因。这一问题与控件的刷新有关。

错误监视

这一问题与使用 `OnDraw()` 方法绘制屏幕的程序员遇到的一个问题类似。`OnDraw()` 方法由于直到处理 `WM_PAINT` 消息才刷新屏幕而臭名昭著。这一消息可以用许多方式产生，包括调整屏幕大小和调用函数 `InvalidateRect()`。

此处的问题当然是，这是一个控件。我们需要一个手段，每当一个“命中”发生时就刷新一次屏幕。



当设置 `m_nSides` 为等于 `newVal` 时，边数的变化反映到方法 `put_Sides()` 中。这是调用 `FireViewChange()` 的理想位置，图 16-17 显示调试过程中改变后的代码。



这一问题确实需要我们知道一些与控件刷新方法有关的内容，而这在开发 ATL COM 组件时遇到的问题中也是很典型的。

和前面一章那样，本章分成两个主要部分。第一部分给出了开发一个基于 Microsoft 的 Polygon 教学例子的 ATL COM 控件的步骤。

— 484 —

第 17 章

STL 和 MFC 编程





在第 12 章和第 13 章中，学习了如何将 STL 的强大功能加入命令行式的 C++ 应用程序中。本章将展示 STL 和 MFC 库如何在一个实用 Windows 应用程序中结合到一起。

在本章中，将学习如何利用 STL<complex>模板开发一个应用程序，同时使用先前在第 10 章和第 11 章中开发的 MFC 工具，然后说明如何在此环境下使用各种调试技巧解决出现的程序问题。

如果读者是一位数学、物理工作者或电气工程师，将熟悉这种应用，因为其中含有复数运算。

17.1 产生一个 STL 和 MFC 应用程序

在这一节开发的应用程序中，单个的复数将和其一起绘制到屏幕上。这些复数利用了 STL<complex>模板和 MFC。首先做一个小小的回顾，以更好地理解复数的实际意义。

17.1.1 复数

在数学、工程和物理领域最频繁地遇到复数。复数来自具有幅值和方向的向量(vector)或旋转向量(phasor)，可以在 x-y 坐标系中描述。从现在起，我们将使用术语旋转向量，以使这一概念不会与 STL<vector>类相混淆。

复数或旋转向量可以用三种形式表示：极坐标形式、直角坐标形式和指数形式。例如，考虑一个幅值为 40、相角为 60 度(自 x 正半轴逆时针测量)的旋转向量的表示。

指数形式：

以指数形式表示，此旋转向量为：

$$40 * e^{j60}$$

极坐标形式：

以极坐标形式表示，此旋转向量可以表示为：

$$40 / _60deg$$

直角坐标形式：

以直角坐标形式表示，实部为：

$$40 * \cos 60 = 40 * 0.5 = 20.0$$

虚部为：

$$40 * \sin 60 = 40 * 0.866025 = 34.6410$$

此旋转向量可表示为：

20.0+j34.6410

旋转向量被称为复数是因为其由实部和虚部两个部分复合而成。

设计提示

复数的虚部前有一个字母。数学家喜欢用字母 i 。然而由于 i 表示电流，物理学家和电气工程师采用了字母 j ，以避免混淆。

由于大多数数学运算符没有重载复数运算，所以计算机中含有复数的数学运算一直都很困难。例如，如果运算符 $+$ 、 $-$ 、 \times 、 $/$ 没有为复数运算重载，那么对复数执行加($+$)、减($-$)、乘(\times)、除($/$)运算是不可可能的，由一个模板处理这些工作也是不可可能的。

读者可能由以前关于复数的工作想起，当这些数以直角坐标形式表示时，它们很容易执行加法和减法运算。例如：

$$\begin{aligned}(60+j40) - (30+j20) &= 30+j20 \\ (20+j45) + (15-j20) &= 35+j25\end{aligned}$$

当复数以极坐标形式表示时，乘法和除法最容易执行，例如：

$$\begin{aligned}(45.0/_63.44\text{deg}) * (-26.05/_{-33.7}\text{deg}) &= (45.0 * -26.05) / _{(63.44 (+) - 33.7)\text{deg}} \\ &= 1172.25/_{29.74}\text{deg} \\ (65/_{50}\text{deg}) / (20/_{-30}\text{deg}) &= (65/20) / _{(50\text{deg} (-) - 30\text{deg})} = 3.25/_{80}\text{deg}\end{aligned}$$

因为加法和减法以直角坐标形式很容易完成而乘法和除法最好地利用了极坐标形式，所以直角坐标形式和极坐标形式之间的转换在含有复数的运算中是很典型的。

在下一节中，将查看 `<complex>` 模板的语法。然后利用 `<complex>` 模板和 MFC 建立一个 Windows 应用程序，在窗口中画出几个旋转向量及其和。

17.1.2 模板语法

标准 C++ 头文件 `<complex>` 用来定义模板类 `complex` 和多个支持模板的函数。下面的清单可以列出了 `<complex>` 模板的语法：

```
namespace std {
#define __STD_COMPLEX
//      TEMPLATE CLASSES
template<class T>
    class complex;
class "complex<float>;
class "complex<double>;
class "complex<long double>;
```



```
//      TEMPLATE FUNCTIONS
template<class T>
    complex<T> operator+(const complex<T>& lhs,
                        const complex<T>& rhs);

template<class T>
    complex<T> operator+(const complex<T>& lhs,
                        const T& rhs);

template<class T>
    complex<T> operator+(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs,
                        const complex<T>& rhs);

template<class T>
    complex<T> operator-(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator-(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator*(const complex<T>& lhs,
                        const complex<T>& rhs);

template<class T>
    complex<T> operator*(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator*(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator/(const complex<T>& lhs,
                        const complex<T>& rhs);

template<class T>
    complex<T> operator/(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator/(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs);
```



```

template<class T>
    bool operator==(const complex<T>& lhs,
                    const complex<T>& rhs);

template<class T>
    bool operator==(const complex<T>& lhs, const T& rhs);
template<class T>
    bool operator==(const T& lhs, const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs,
                    const complex<T>& rhs);

template<class T>
    bool operator!=(const complex<T>& lhs, const T& rhs);
template<class T>
    bool operator!=(const T& lhs, const complex<T>& rhs);
template<class E, class Ti, class T>
    basic_istream<E, Ti>& "operator">>(basic_istream<E, Ti>& is,
                                        complex<T>& x);

template<class E, class T, class U>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os,
                                    const complex<U>& x);

template<class T>
    T real(const complex<T>& x);
template<class T>
    T imag(const complex<T>& x);
template<class T>
    T abs(const complex<T>& x);
template<class T>
    T arg(const complex<T>& x);
template<class T>
    T norm(const complex<T>& x);
template<class T>
    complex<T> conjg(const complex<T>& x);
template<class T>
    complex<T> polar(const T& rho, const T& theta = 0);
    
```



```
template<class T>
    complex<T> cos(const complex<T>& x);
template<class T>
    complex<T> cosh(const complex<T>& x);
template<class T>
    complex<T> exp(const complex<T>& x);
template<class T>
    complex<T> log(const complex<T>& x);
template<class T>
    complex<T> log10(const complex<T>& x);
template<class T>
    complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x, const T& y);
template<class T>
    complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T>
    complex<T> pow(const T& x, const complex<T>& y);
template<class T>
    complex<T> sin(const complex<T>& x);
template<class T>
    complex<T> sinh(const complex<T>& x);
template<class T>
    complex<T> sqrt(const complex<T>& x);
};
```

为这一模板类返回多个值的函数将返回一个在半开区间 $(-\pi, \pi]$ 内的虚部。

错误监视

在本书中，求复数的复共轭要用 `conj()`，而不是 `conjg()`，参见 Microsoft 参考手册。

表 17-1 描述了 `<complex>` 的模板函数。

表 17-1 <complex>的模板函数

模板
<code>complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);</code>
<code>complex<T> operator+(const complex<T>& lhs, const T& rhs);</code>
<code>complex<T> operator+(const T& lhs, const complex<T>& rhs);</code>
<code>complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);</code>
<code>complex<T> operator-(const complex<T>& lhs, const T& rhs);</code>
<code>complex<T> operator-(const T& lhs, const complex<T>& rhs);</code>
<code>complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);</code>
<code>complex<T> operator*(const complex<T>& lhs, const T& rhs);</code>
<code>complex<T> operator*(const T& lhs, const complex<T>& rhs);</code>
<code>complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);</code>
<code>complex<T> operator/(const complex<T>& lhs, const T& rhs);</code>
<code>complex<T> operator/(const T& lhs, const complex<T>& rhs);</code>
<code>complex<T> operator+(const complex<T>& lhs);</code>
<code>complex<T> operator-(const complex<T>& lhs);</code>
<code>Bool operator==(const complex<T>& lhs, const complex<T>& rhs);</code>
<code>Bool operator==(const complex<T>& lhs, const T& rhs);</code>
<code>Bool operator==(const T& lhs, const complex<T>& rhs);</code>
<code>Bool operator!=(const complex<T>& lhs, const complex<T>& rhs);</code>
<code>Bool operator!=(const complex<T>& lhs, const T& rhs);</code>
<code>Bool operator!=(const T& lhs, const complex<T>& rhs);</code>
<code>basic_istream<E, Ti>& operator>>(basic_istream<E, Ti>& is, complex<T>& x);</code>
<code>basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const complex<U>& x);</code>

表 17-2 描述了这一模板类中的方法。注意这一模板提供所有处理复数所需的标准操作。

表 17-2 各种<complex>模板方法

模板方法	描述
<code>template<class T> T abs(const complex<T>& x);</code>	返回 x 的绝对值
<code>template<class T> T arg(const complex<T>& x);</code>	返回 x 的相角
<code>template<class T> complex<T> conjg(const complex<T>& x);</code>	返回 x 的共轭 注：此时使用 conj() 获得复数共轭
<code>template<class T> complex<T> cos(const complex<T>& x);</code>	返回 x 的余弦
<code>template<class T> complex<T> cosh(const complex<T>& x);</code>	返回 x 的双曲余弦
<code>template<class T> complex<T> exp(const complex<T>& x);</code>	返回 x 的幂
<code>template<class T> T imag(const complex<T>& x);</code>	返回 x 的虚数部分
<code>template<class T> complex<T> log(const complex<T>& x);</code>	返回 x 的对数。分支切割于负实轴



(续表)

模板方法	描述
template<class T> complex<T>log10(const complex<T>& x);	返回 x 以 10 为底的对数。分支切割于负实轴
template<class T> T norm(const complex<T>& x);	返回 x 的平方值
template<class T> complex<T>polar(const T& rho, const T& theta = 0);	返回一个复数值，绝对值为 rho，相角为 theta
template<class T> complex<T>pow(const complex<T>& x, int y); template<class T> complex<T>pow(const complex<T>& x, const T& y); template<class T> complex<T>pow(const complex<T>& x, const complex<T>& y); template<class T> complex<T>pow(const T& x, const complex<T>& y);	每一个函数都将两个操作对象转换为给定的返回类型，然后返回转换后 x 的 y 次幂。对于 x 的分支切割发生于负实轴
template<class T> T real(const complex<T>& x);	返回 x 的实数部分
template<class T> complex<T>sin(const complex<T>& x);	返回 x 的虚数正弦
template<class T> complex<T>sinh(const complex<T>& x);	返回 x 的双曲正弦
template<class T> complex<T>sqrt(const complex<T>& x);	返回 x 的平方根，相角位于半开区间 $(-\pi/2, \pi/2]$ ，分支切割发生于负实轴

<complex>模板类描述了一个对象。此对象中保存了类型 T 的两个对象。其中一个对象表示复数的实部，而另一个对象表示复数的虚部。

类 T 的对象有一个公共的构造函数、析构函数、拷贝构造函数和赋值运算符，类 T 的对象可被赋值为整型或浮点型数值，或者转换为期望的数值。为适当的浮点类型定义了算术运算符。

模板类处理三种浮点类型：float、double 和 long double，对于这个版本的 Visual C++，在实际计算中，任何其他类型的 T 均被转换为 double 型。返回的类型 double 被赋给类型 T 的对象。

类 complex<float>描述一个保存二 float 型对象的对象。其中一个 float 型对象表示复数的实部，而第二个表示复数的虚部。

同样，类 complex<double>描述一个保存二个 double 型对象的对象，一个对象表示复数的实部，而第二个对象表示复数的虚部。

最后，类 complex<long double>描述一个保存二 long double 型对象的对象。一个对象表示复数的实部，而第二个对象表示复数的虚部。

17.1.3 基本的应用程序代码

要建立本章的工程，需按照第 10 章概括的利用 AppWizard 开发 MFC 应用程序的步骤执行，命名工程为“Vectors”。

错误监视

在代码中，STL 允许指定 `<complex>` 数为直角坐标形式或极坐标形式。然而，在 Watch 窗口中查看结果时，这些结果将以直角坐标形式显示，实部和虚部分开。

当 AppWizard 已经为应用程序生成了源代码和头文件时，选择源代码文件 `VectorsView.cpp` 并作修改，在其中添加下面清单中以黑体显示的代码：

```
// VectorsView.cpp : implementation of the CVectorsView class.
//
#include "stdafx.h"
#include "Vectors.h"
#include "VectorsDoc.h"
#include "VectorsView.h"
#include <complex>
using namespace std;
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CVectorsView
IMPLEMENT_DYNCREATE(CVectorsView, CView)
BEGIN_MESSAGE_MAP(CVectorsView, CView)
    //{{AFX_MSG_MAP(CVectorsView)
    ON_WM_SIZE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CVectorsView construction/destruction
CVectorsView::CVectorsView()
{
}
CVectorsView::~CVectorsView()
{
}
BOOL CVectorsView::PreCreateWindow(CREATESTRUCT& cs)
```



```
{
    return CView::PreCreateWindow(cs);
}
////////////////////////////////////
// CVectorsView drawing
void CVectorsView::OnDraw(CDC* pDC)
{
    CVectorsDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CPen bluepen, greenpen, yellowpen, redpen;
    CPen* oldpen;
    complex<double> x1, x2, x3, temp;
    // Hard wire the three phasor values
    // phasor one
    x2.real(-85.5);           //-85.5 - j55.2
    x2.imag(-55.2);
    // phasor two
    x1 = polar(63.7, -0.523598); //63.7 / -30 deg
    // phasor three
    x3.real(-75.0)           //-75.0 + j117.9
    x3.imag(117.9);
    // set mapping modes and viewport
    pDC->SetMapMode(MM_ISOTROPIC);
    pDC->SetWindowExt(300,300);
    pDC->SetViewportExt(m_cxClient,-m_cyClient);
    pDC->SetViewportOrg(m_cxClient/2,m_cyClient/2);
    // draw coordinate axes
    pDC->MoveTo(-150,0);
    pDC->LineTo(150,0);
    pDC->MoveTo(0,-125);
    pDC->LineTo(0,125);
    // draw first phasor with green pen
    greenpen.CreatePen(PS_SOLID,2,RGB(0,255,0));
    oldpen = pDC->SelectObject(&greenpen);
    pDC->MoveTo(0,0);
    pDC->LineTo(real(x1),imag(x1));
    DeleteObject(oldpen);
    temp = x1 + x2; // add first two phasors
    // draw second phasor with blue pen
    bluepen.CreatePen(PS_SOLID,2,RGB(0,0,255));
    oldpen = pDC->SelectObject(&bluepen);
    pDC->MoveTo(0,0);
    pDC->LineTo(real(x2),imag(x2));
```

```

DeleteObject(oldpen);
temp += x3;    // add in last phasor
// draw third phasor with yellow pen
yellowpen.CreatePen(PS_SOLID,2,RGB(255,255,0));
oldpen = pDC->SelectObject(&yellowpen);
pDC->MoveTo(0,0);
pDC->LineTo(real(x3),imag(x3));
DeleteObject(oldpen);
// draw sum of phasors with wide red pen
redpen.CreatePen(PS_SOLID,4,RGB(255,0,0));
oldpen = pDC->SelectObject(&redpen);
pDC->MoveTo(0,0);
pDC->LineTo(real(temp),imag(temp));
DeleteObject(oldpen);
}
////////////////////////////////////
// CVectorsView diagnostics
#ifdef _DEBUG
void CVectorsView::AssertValid() const
{
    CView::AssertValid();
}
void CVectorsView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}
CVectorsDoc* CVectorsView::GetDocument() // non-debug is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CVectorsDoc)));
    return (CVectorsDoc*)m_pDocument;
}
#endif // _DEBUG
////////////////////////////////////
// CVectorsView message handlers
void CVectorsView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    // TODO: Add your message handler code here
    m_cxClient = cx;
    m_cyClient = cy;
}

```



要完成此应用程序，还有几步是必要的。首先，从 Visual C++ 编译器的 View 菜单中打开 ClassWizard。将所有窗口设为图 17-1 中的值，然后添加一个 WM_SIZE 消息处理程序。

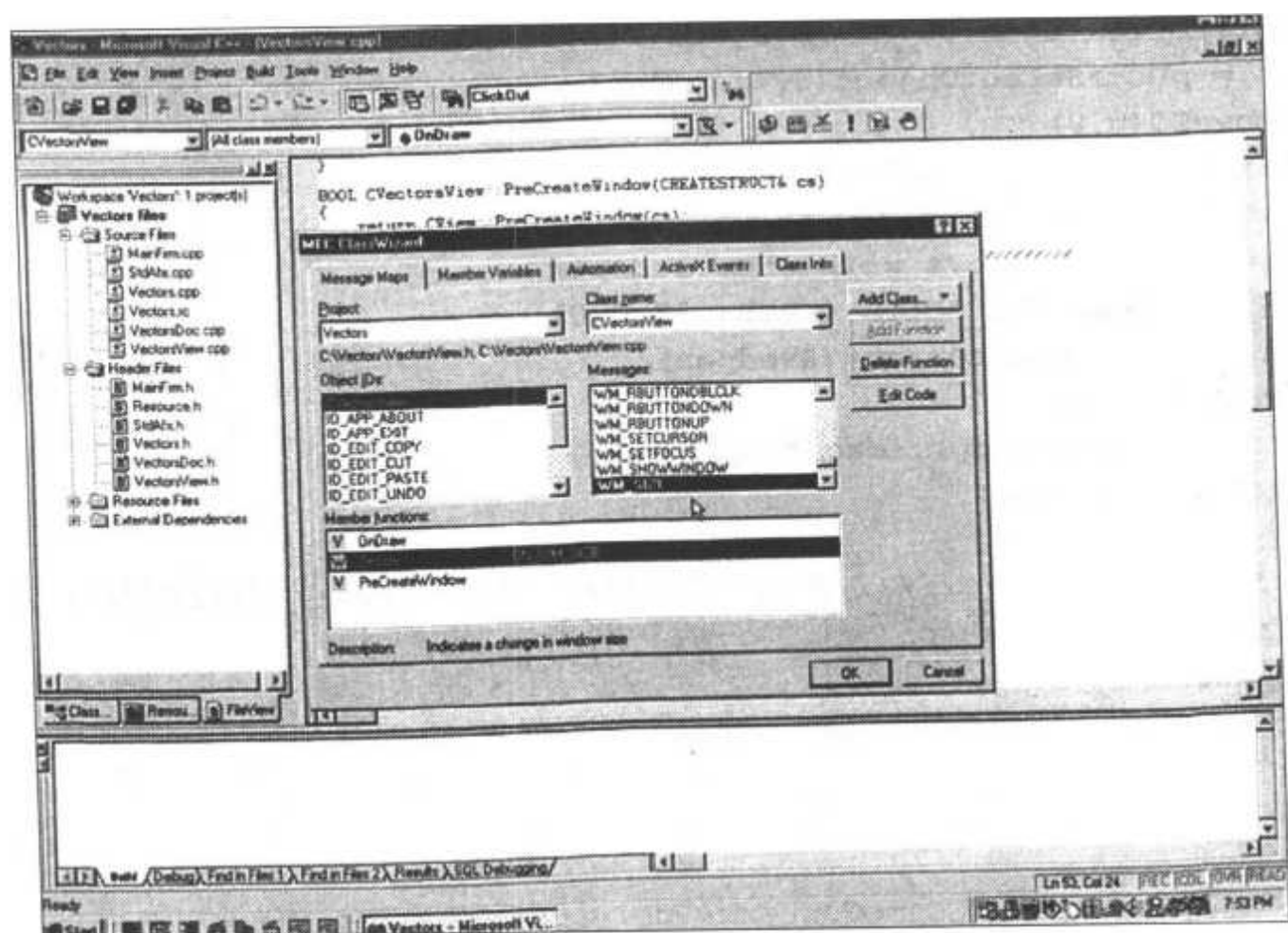


图 17-1 可以使用 ClassWizard 添加 WM_SIZE 消息处理程序

ClassWizard 将在源代码文件 VectorsView.cpp 中添加一个 OnSize() 方法，如图 17-2 所示。修改代码，将有关窗口大小的数据赋值给两个成员变量 m_cxClient 和 m_cyClient。

```
void CVectorsView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Add your message handler code here
    m_cxClient = cx;
    m_cyClient = cy;
}
```

为使这些变量可见，在头文件 VectorsView.h 中声明这些成员变量。下面的清单以黑体字显示了必须在此头文件中输入的实际代码：

```
class CVectorsView : public CView
{
    private:    // member variables for resized window
    int m_cxClient;
    int m_cyClient;
    protected: // create from serialization only
```



```

CVectorsView();
DECLARE_DYNCREATE(CVectorsView)
// Attributes
public:
    CVectorsDoc* GetDocument();
    .
    .
    .

```

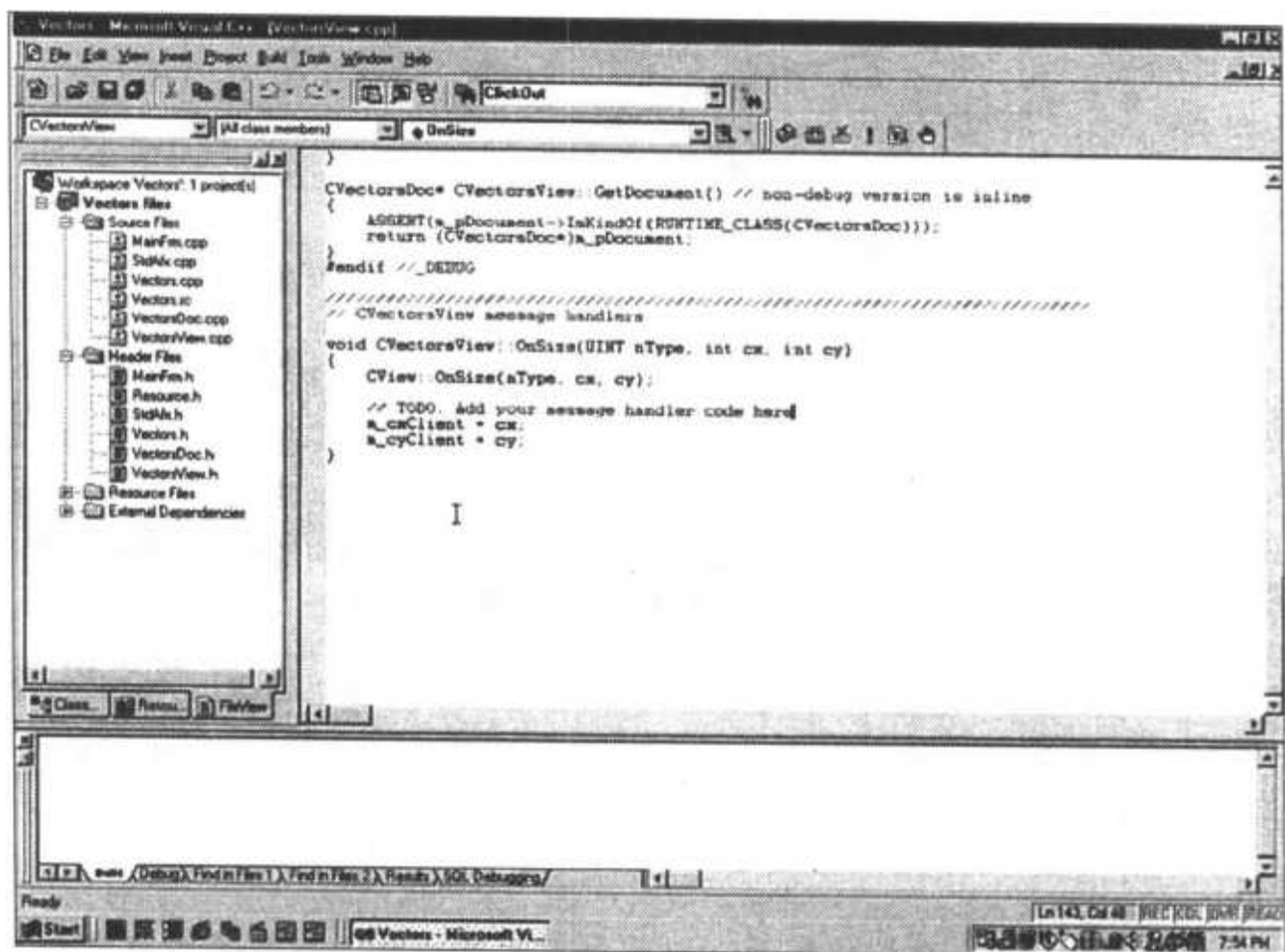


图 17-2 通过 ClassWizard 在工程中添加了一个 OnSize()方法

剩下唯一要做的就是编译和执行这一应用程序了，图 17-3 显示了窗口中的输出。

原始的三个旋转向量以细线段显示，这三个旋转向量相加的和以粗红线显示。现在旋转向量(复数)的加法可行了，因为运算符“+”被重载。

这时，我们还不知道结果是否正确。为确认，不得不转向 Debugger 并建立一个 Watch 窗口。另外，如果将这些旋转向量头尾相接，而不是都以(0,0)为起点，结果将一目了然。

17.2 调试

因为我们只对 Watch 窗口中查看复数的和感兴趣，所以这一次不必使用远程调试。然而，如果愿意远程调试，可以按照原来在第 8 章中概括的步骤执行。

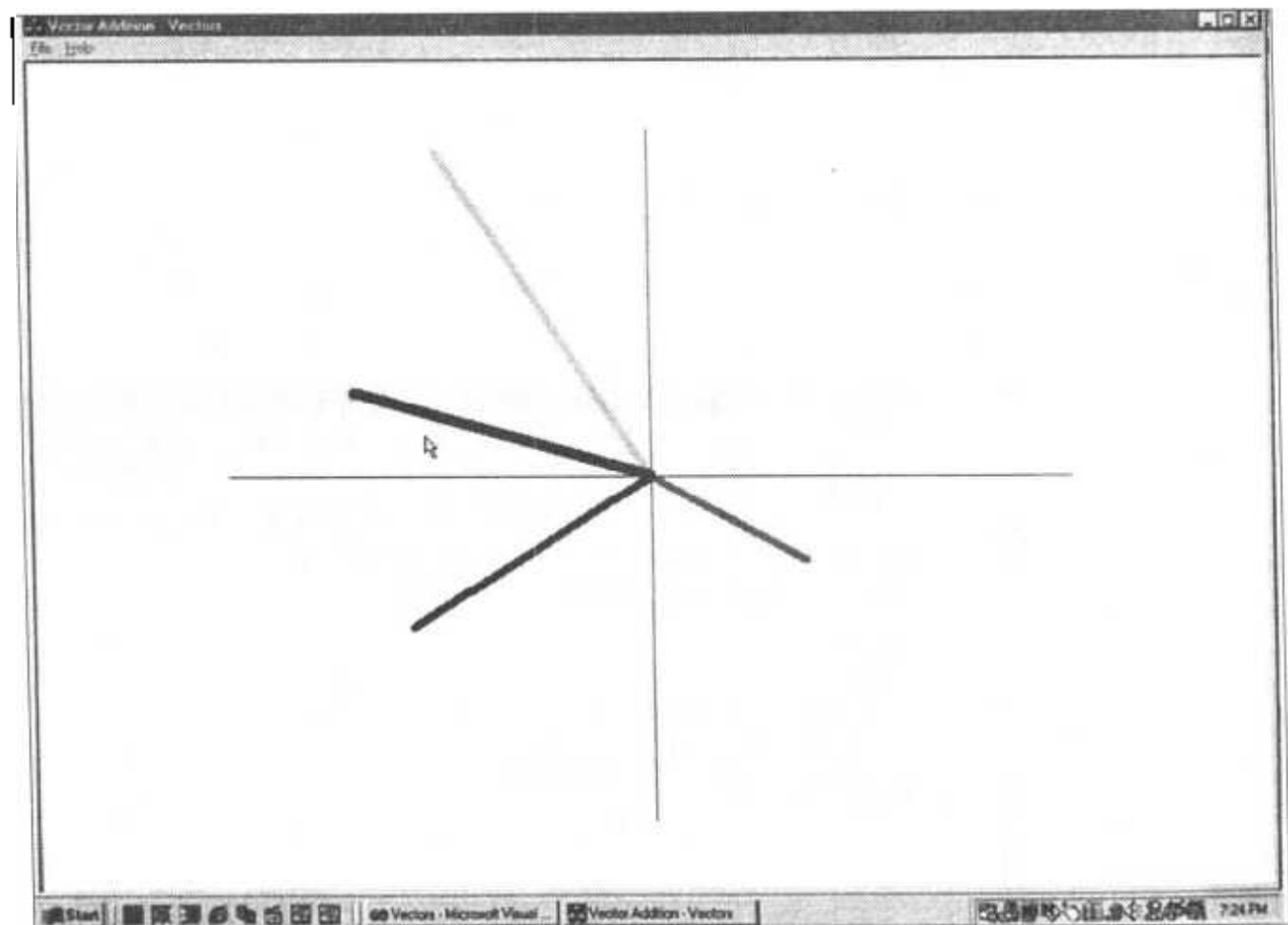


图 17-3 窗口中复数计算的图形结果

变量 `temp` 是 `<complex>` 类型的，因此需要将其和三个旋转向量 `x1`、`x2` 及 `x3` 一起输入到一个 Watch 窗口。在 `x1` 和 `x2` 被添加方法 `OnDraw()` 之后，立即设一个断点。在 `x3` 被加进和值后，设第二个断点。

设计提示

在图 17-4 中，每一个复数均被展开，以观察实部和虚部。这样复数就分解为实部和虚部两个部分。

我们立即注意到，在图 17-4 中复数的信息是以直角坐标形式返回的。每一个复数都表达为实部加上虚部。在图 17-4 中，只有 `x1` 和 `x2` 相加。结果正确否？

```
+55.165842923858-j31.849957213476  
-85.500000000000-j55.200000000000  
-----  
-30.334157076142-j87.049957213476
```

是的，这一中间结果是正确的。下面执行最后的加法。图 17-5 显示了应用程序执行到第二个断点后的 Watch 窗口。

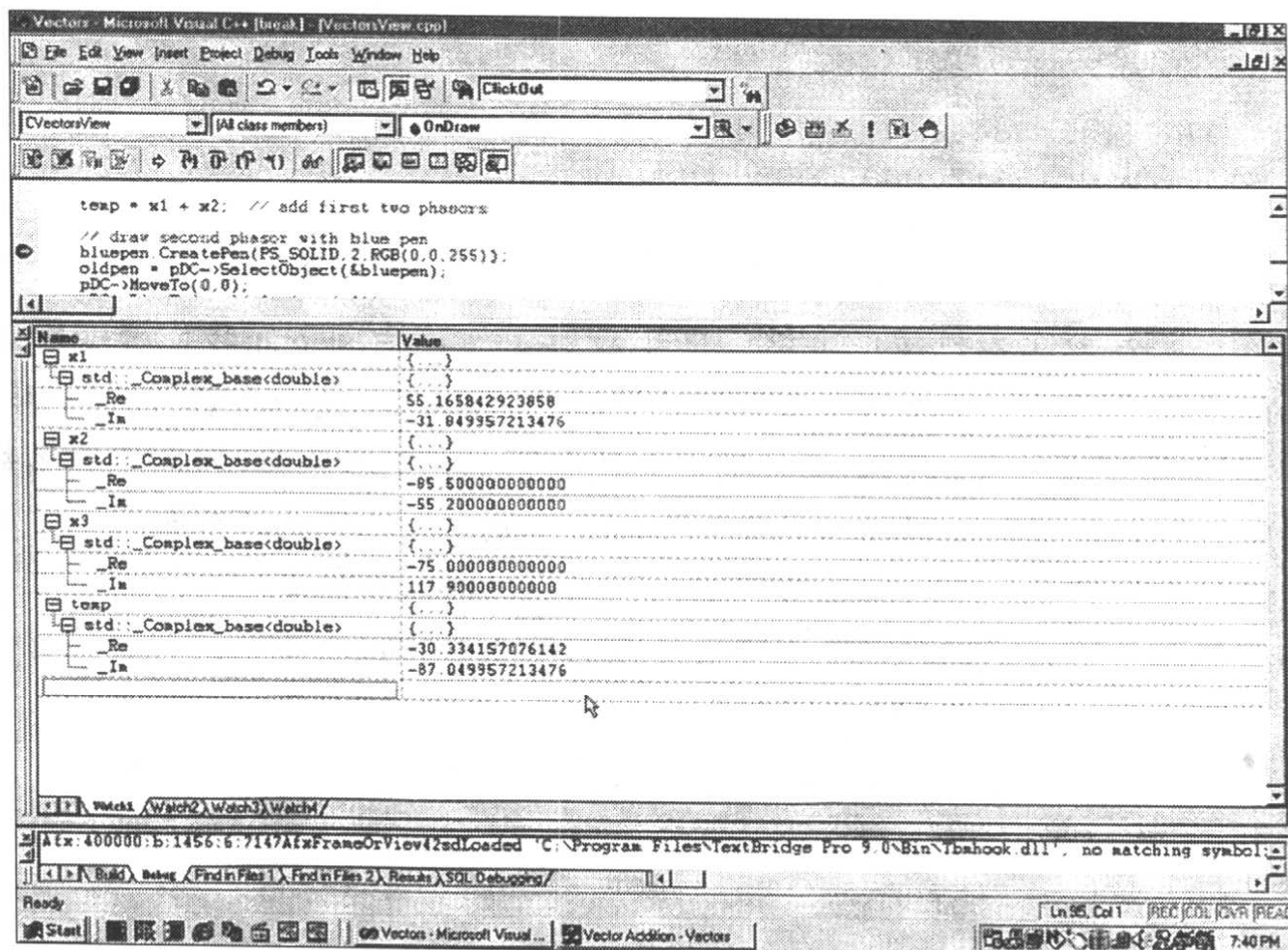


图 17-4 原始复数和中间和

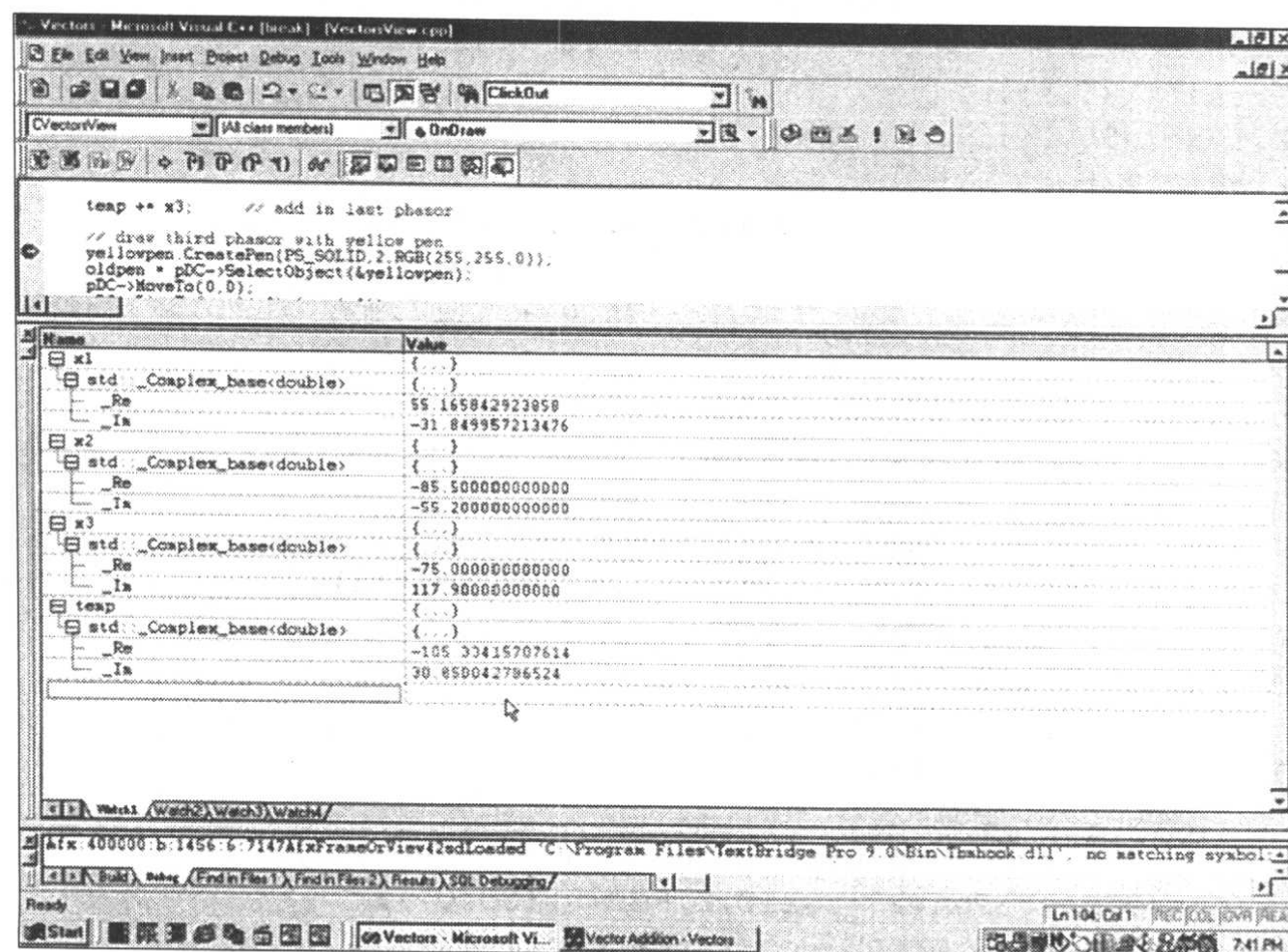


图 17-5 最终的加法完成后，三个复数的和被显示



再执行一次计算，检验结果：

```
-30.334157076142-j87.049957213476  
-75.000000000000+j117.900000000000  
-----  
-105.33415707614+j30.850042786524
```

再一次，我们得到了对结果的证实。应用程序工作正确。下面我们改变旋转向量被画出的样子。现在，窗口中每个旋转向量的起始点均为点(0,0)。我们希望将这些旋转向量画成头尾相接的样子。要从第一个旋转向量的尖端开始画第二个旋转向量，只需删除函数 MoveTo() 调用，并将此旋转向量画至变量 temp 中的临时和值处。用同样的过程画第三个旋转向量。当需要画和值时，从第三个旋转向量的尖端向起点(0,0)画一条宽线。

下面的清单显示了为完成这些变化需要修改的方法 OnDraw()：

```
void CVectorsView::OnDraw(CDC* Pdc)  
{  
    CVectorsDoc* pDoc = GetDocument();  
    ASSERT_VALID(pDoc);  
    CPen bluepen, greenpen, yellowpen, redpen;  
    CPen* oldpen;  
    complex<double> x1, x2, x3, temp;  
    // Hard wire the three phasor values  
    // phasor one                //-85.5 - j55.2  
    x2.real(-85.5);  
    x2.imag(-55.2);  
    // phasor two  
    x1 = polar(63.7, -0.523598); //63.7 /_ -30 deg  
    // phasor three  
    x3.real(-75.0);              //-75.0 + j117.9  
    x3.imag(117.9);  
    // set mapping modes and viewport  
    pDC->SetMapMode(MM_ISOTROPIC);  
    pDC->SetWindowExt(300,300);  
    pDC->SetViewportExt(m_cxClient,-m_cyClient);  
    pDC->SetViewportOrg(m_cxClient/2,m_cyClient/2);  
    // draw coordinate axes  
    pDC->MoveTo(-150,0);  
    pDC->LineTo(150,0);  
    pDC->MoveTo(0,-152);  
    pDC->LineTo(0,125);  
    // draw first phasor with green pen  
    greenpen.CreatePen(PS_SOLID,2,RGB(0,255,0));  
    oldpen = pDC->SelectObject(&greenpen);  
    pDC->MoveTo(0,0);
```

```
pDC->LineTo(real(x1), imag(x1));
DeleteObject(oldpen);
temp = x1 + x2; // add first two phasors
// draw second phasor with blue pen
bluepen.CreatePen(PS_SOLID, 2, RGB(0, 0, 255));
oldpen = pDC->SelectObject(&bluepen);
pDC->LineTo(real(temp), imag(temp));
DeleteObject(oldpen);
temp += x3; // add in last phasor
// draw third phasor with yellow pen
yellowpen.CreatePen(PS_SOLID, 2, RGB(255, 255, 0));
oldpen = pDC->SelectObject(&yellowpen);
pDC->LineTo(real(temp), imag(temp));
DeleteObject(oldpen);
// draw sum of phasors with wide red pen
redpen.CreatePen(PS_SOLID, 4, RGB(255, 0, 0));
oldpen = pDC->SelectObject(&redpen);
pDC->LineTo(0, 0);
DeleteObject(oldpen);
}
```

图 17-6 显示了在窗口中新画的图。

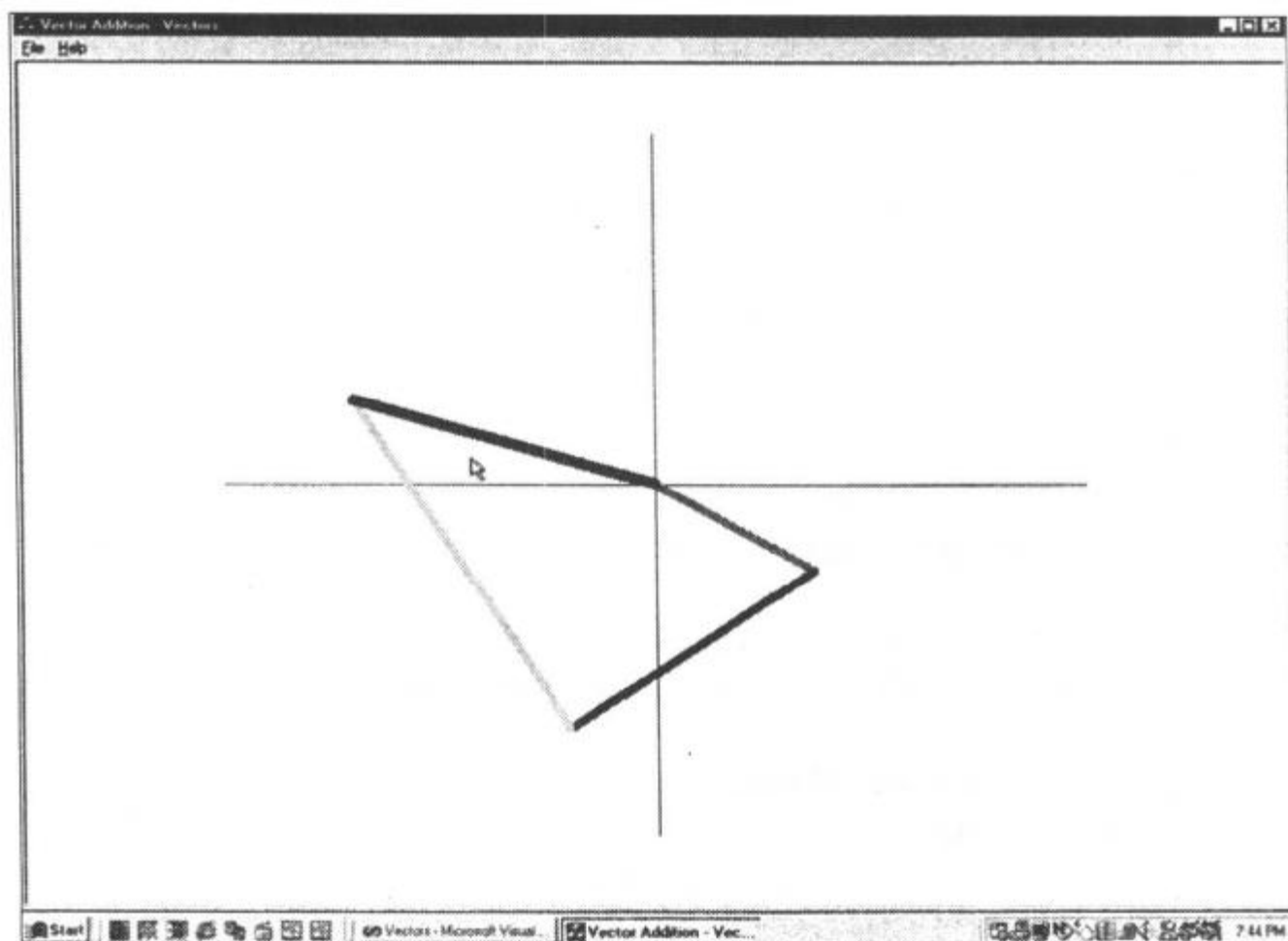


图 17-6 修改方法 OnDraw() 改变了图形的外观



注意画出的三个旋转向量是头尾相接的，鼠标指针正指向表示三个原始旋转向量 x_1 、 x_2 和 x_3 的和值的红线。

24x7 可靠的 STL 编码和调试实践

用 STL 和 MFC 编写的可靠代码，远非本章的例子 Vectors 可比，应用程序的使用范围和 STL 以及 MFC 库自身一样可靠而种类繁多。

还记得我们在第 11 章介绍的一个用 MFC 编写的名叫 Fourier 的应用程序否？那个 Fourier 在 MFC 环境下使用通常的 C++ 代码，在窗口中绘制出一傅立叶波形。那个程序是一个用 STL 实现的好例子。首先，回到第 11 章，回顾该章示例 Fourier 的编程思想。现在，查看下面这段修改过的在方法 FourierView::OnDraw() 中使用的代码。要特别注意以黑体字显示的代码。

```
.
.
.
#include <numeric>
#include <vector>
#include <math.h>
using namespace std;
typedef vector <float> FourierArray;
.
.
.
////////////////////////////////////
// CFourierView drawing
void CFourierView::OnDraw(CDC* pDC)
{
    CFourierDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int ltitle;
    double y, yp;
    double vertscale, horzscale;
    //vertical plotting scaling factor
    vertscale = 180.0;
    // convert degrees to radians and scale
    // horizontal for 360 degrees in 400 points
    horzscale = 3.1415927 * 360 / (180 * 400);
    // define a vector of floats
    FourierArray rgFA;
    // set mapping mode, viewport, and so on
    pDC->SetMapMode(MM_ISOTROPIC);
    pDC->SetWindowExt(500,500);
    pDC->SetViewportExt(m_cxClient, -m_cyClient);
```

```

pDC->SetViewportOrg(m_cxClient/20,m_cyClient/2);
// draw x & y coordinate axes
pDC->MoveTo(0,240);
pDC->LineTo(0,-240);
pDC->MoveTo(0,0);
pDC->LineTo(400,0);
pDC->MoveTo(0,0);
// i represents a given angle for the series
for (int i = 0; i <= 400; i++) {
    // calculate Fourier terms for the angle
    // place each term in the array
    for (int j=1; j<=pDoc->myterms; j++) {
        y = (vertscales / ((2.0 * j) - 1.0)) * \
            sin(((j * 2.0) - 1.0) * horzscales * i);
        rgFA.push_back(y);
    }
    // accumulate the individual array terms
    // for the angle
    yp = accumulate(rgFA.begin(),rgFA.end(),0.0f);
    // draw the scaled point in the client area
    pDC->LineTo(i, (int)yp);
    yp-=yp;
    // clean out the array and prepare
    // with next angle's values
    for (j=1; j<=pDoc->myterms; j++) {
        rgFA.pop_back();
    }
}
// print waveform title
ltitle=strlen(pDoc->mytext);
pDC->TextOut(200-(ltitle*8/2),200,pDoc->mytext,ltitle);
}
.
s.
.

```

回想在第 11 章中，为每一个画到屏幕上的点，傅立叶级数的每一项分别用程序计算。利用向量成员函数 `push_back()` 将各项值存入一个数组。对于给定的每一个角度，每一个谐波项计算结果的和是由成员函数 `accumulate()` 累加数组中的每个值求得的。在处理下一个角度之前，利用成员函数 `pop_back()` 将每一项的值从数组中删除。

注意下面这行位于程序清单开始处的代码：

```
typedef vector <float> FourierArray ;
```



关键字 `float(<float>)` 外边的尖括号指示编译器以标准 `float` 类型填充所有用向量模板化的数据类型的定义，这种类型定义用于实际向量的实例化：

```
FourierArray rgFA ;
```

实际的 Fourier 计算是在两个 `for` 循环中执行的，Fourier 图用函数 `LineTo()` 绘制。

外层 `for` 循环用下标 `i` 递增窗口中绘制点的水平位置。这一数值表示为一组傅立叶级数项的角度值。

内层 `for` 循环用下标 `j` 给出对于指定角度所计算的傅立叶级数谐波项数量。例如，如果 `i` 指向一个代表 45 度，傅立叶谐波项为 10，那么在内层循环中将对每个 `i` (例如，45 度) 执行 10 次计算，计算结果放入数组中。这是利用 STL 动态数组即向量方法实现的，利用以下语句插入元素：

```
rgFA.push_back(y) ;
```

这样，对于给定角度，每一个傅立叶项都在此数组中累加。各项值可以用方法 `accumulate()` 累加组成一个绘图数据点。这一方法不是向量模板的一部分，但是在 STL 数组型模板中定义了。方法 `accumulate()` 用以下方式使用：

```
y = accumulate(rgFA.begin(), rgFA.end(), 0.0f) ;
```

数值型方法 `accumulate()` 以初值 `rgFA.begin()` 初始化累加器 `yp`，然后对此范围中的每个向量项用 `yp=yp+*double_ptr` 依次修改，直到 `rgFA.end()`。通常，方法 `accumulate()` 用来累加向量中的数值型元素，需要一个指定容器的迭代器(iterator)来确定求和的范围。这些迭代器(即指针值)由方法 `begin()` 和 `end()` 提供。

最后，用一个类似的 `for` 循环控制语句清除向量：

```
for (j=1; j<=pDoc->myterms; j++) {  
    rgFA.pop_back() ;  
}
```

向量方法 `pop_back()` 从列表中删除最后一个元素。也可以调用向量方法 `clear()`，它依次调用方法 `erase()` 和 `end()` 从头到尾自动删除向量的内容。

通过 Debugger 的 Watch 窗口，可以看到元素放入数组的情况。图 17-7 显示跳过几次 `for` 循环后的 Watch 窗口中的 `rgFA` 和 `first`、`last` 和 `end` 元素。

观察对 STL 数组执行压入和弹出操作的功能为 STL 和 MFC 应用程序提供了实实在在的调试功能。

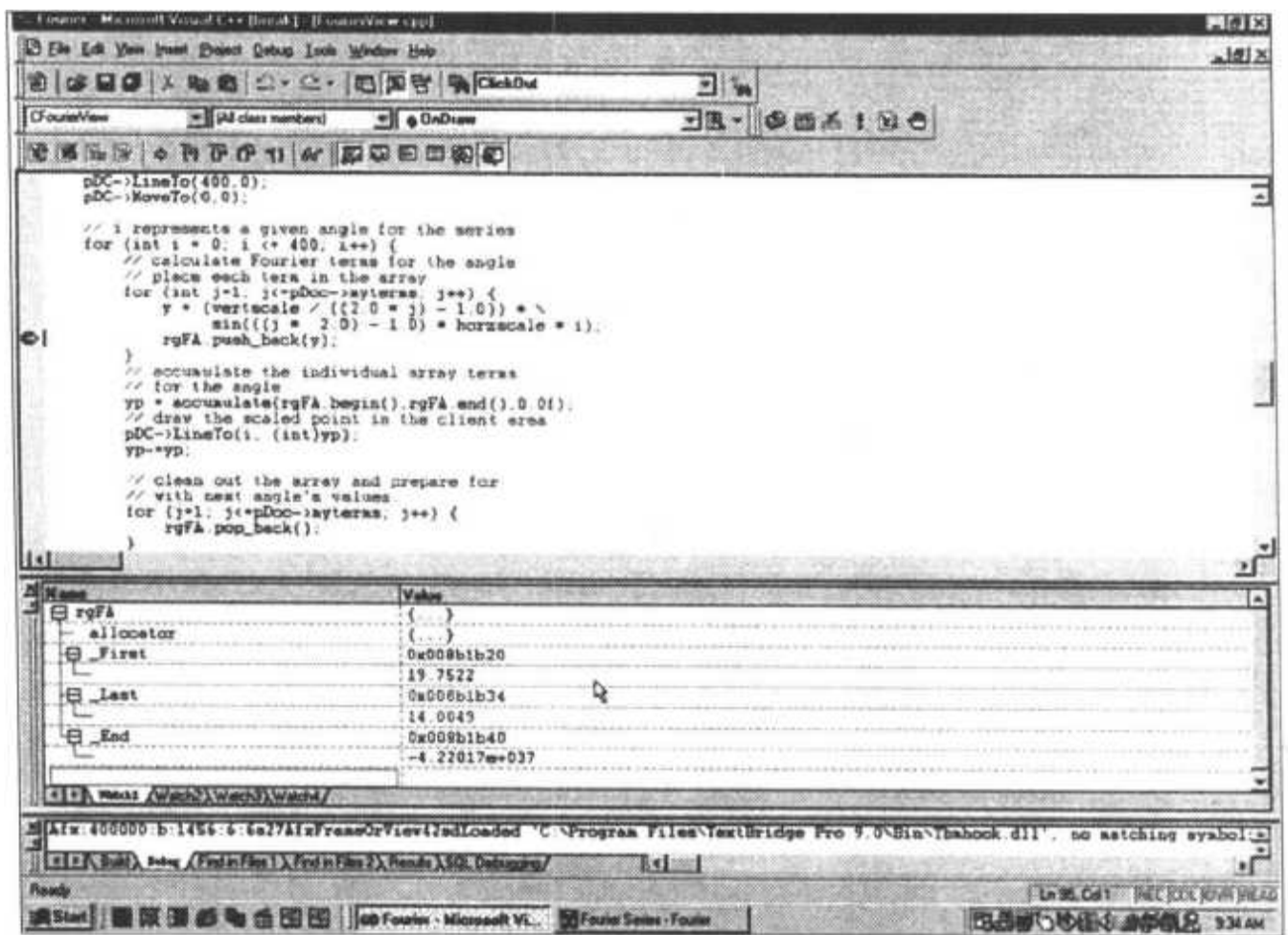


图 17-7 在 Watch 窗口中查看 rgFA

17.3 小结

在本章中，学习了如何在 Microsoft Foundation Class(MFC)环境下使用 Standard Template Library(STL)。结合 STL 和 MFC 开发的 Vectors 程序代码依赖于前几章打下的基础，比如第 11 章到第 13 章。

本章还学习了在 Watch 窗口中，Debugger 可以保存诸如 `STL<complex>` 复数这样的值，并且查看直角坐标形式复数的实部和虚部。利用 `<numeric>` 和 `<vectors>`，学习了当 STL 数组的元素压入和弹出数组时，如何查看它们。 ■

[G e n e r a l I n f o r m a t i o n]

书名= 软件工程师捉虫系列 C + + 程序调试实用手册

作者= B E X P

页数= 5 0 5

下载位置= [http : / / r e a d 3 . 5 r e a d . c o m / d i s k n j s / n j s 2 4 5 / 0 3 / ! 0 0 0 0 1 . p d g](http://read3.5read.com/disknjs/njs245/03/!00001.pdg)

封面
书名
版权
前言
目录
正文